

George Mason University

CDS 230: Modeling and Simulation I

Spring 2020

Dr. Carlos Cruz (main contributor)

Dr. Hamdi Kavak (Spring 2020 contributor)

Course Notes

Contents

1	Setup	3
1.1	The Python Programming Language	3
1.2	Your first program	3
1.2.1	What is a Python program?	4
1.2.2	Hello, world!	4
1.2.3	Executing a Python Program	4
1.2.4	Four Ways to Run Python	5
1.2.5	What to use?	7
1.3	Input and Output	7
1.4	Modules	8
1.5	Intrinsic Functions	8
1.5.1	Optional Arguments	9
1.6	Errors	10
1.7	Programming Style	10
1.8	References	10
2	Variables and Data Types	11
2.1	Fundamentals	11
2.1.1	Operators	11
2.1.2	Summary of Python Arithmetic Operators	11
2.1.3	Python Expressions	11
2.2	Keywords and Identifiers	13
2.3	Variables	13
2.3.1	Assignment statements	13
2.3.2	Namespaces	14
2.4	Built-in Data Types	14
2.4.1	Fundamental Types	14
2.5	Formatting Text and Numbers	19
2.5.1	Number Formatting	20
2.5.2	string.format() basics	20

2.5.3	F-strings	21
2.6	Problem Solving with Python	21
3	Control Flow	27
3.1	Boolean Expressions (Conditions)	27
3.1.1	Conditional Operators	27
3.1.2	Logical Operators	28
3.2	Conditionals	29
3.2.1	if Statements	29
3.3	Loops	32
3.3.1	Indefinite Iteration: while Loops	32
3.3.2	for Loops	34
4	Lists and Tuples	36
4.1	Sequence Types	36
4.2	Lists	36
4.2.1	Initialization	36
4.2.2	Accessing and Editing Lists	37
4.2.3	Slicing	37
4.2.4	Operators	38
4.2.5	Iterating Over List Elements vs. List Index Values	39
4.2.6	List Comprehensions	40
4.2.7	Traversing Multiple Lists Simultaneously	40
4.2.8	Copying Lists	41
4.3	Tuples	41
4.3.1	Why use a tuple instead of a list	42
4.3.2	Nested Data Structures	43
5	Dictionaries and Sets	44
5.1	Sets	44
5.1.1	Properties	44
5.1.2	Initialization	44
5.1.3	Set operations	45
5.1.4	Modifying a Set	45
5.2	Dictionaries	46
5.2.1	Properties	46
5.2.2	Initialization	47
5.2.3	Dictionary Operations	47

6	Functions	50
6.1	Functions	50
6.1.1	Defining Functions	51
6.1.2	Docstrings	51
6.1.3	Value-Returning Functions	52
6.1.4	Non-Value-Returning Functions	52
6.1.5	Calling Functions	53
6.1.6	Parameter Passing	54
6.1.7	Keyword Arguments	54
6.1.8	Default Arguments	55
6.1.9	Variable Scope	55
6.1.10	Functions as Arguments to Functions	56
6.1.11	The Main Program	58
6.1.12	Lambda Functions	59
6.2	A Bioinformatics Example	59
6.2.1	Counting Letters in DNA Strings	60
7	File IO	64
7.1	Reading from a text file	64
7.2	Reading a Mixture of Text and Numbers	67
7.3	With statements	68
7.4	Writing to a text file	69
7.5	Binary Files	70
7.6	More on encoding and decoding: Unicode characters	71
7.6.1	ord() and chr()	72
8	Arrays	74
8.1	Lists as Arrays	74
8.2	2D Arrays	77
8.3	Other Arrays in Python	82
8.3.1	Basic Typed Arrays	82
8.3.2	Immutable Arrays of Unicode Characters	82
8.3.3	Immutable Arrays of Single Bytes	83
8.3.4	Mutable Arrays of Single Bytes	83
9	Numpy	85
9.1	When to Use Numpy	85
9.2	Numpy Data Types	86
9.3	NumPy Array Fundamentals	87

9.3.1	Other ways to create arrays	90
9.4	NumPy Array Operations	91
9.4.1	Basic operations	91
9.4.2	Comparison operators and value testing	93
9.4.3	Array item selection and manipulation	93
9.4.4	Vector and matrix mathematics	94
9.4.5	Statistics	95
9.4.6	Random Numbers	96
9.4.7	Note about operations	97
9.5	Numpy and Text IO	97
10	Basic Plotting	99
10.1	Matplotlib: Pylab	99
10.2	Matplotlib: Pyplot	102
11	Object Oriented Programming	103
11.1	Classes	103
11.1.1	Defining a class	103
11.1.2	Creating objects	105
11.1.3	Accessing Attributes and Methods	106
11.1.4	Data Hiding	108
11.1.5	Passing Objects as Arguments to Function	109
11.2	Inheritance and Polymorphism	110
11.3	Multiple Inheritance	115
11.4	Polymorphism and Method Overriding	116
11.5	Operator Overloading	117
11.6	object : The Base Class (Optional)	120
A	Using the Command Line	122
A.1	Using the Command Line	122
A.1.1	Mac/Linux	122
A.1.2	Windows	123
B	Computational Problem Solving	126
C	References	128

Chapter 1

Setup

1.1 The Python Programming Language

This course provides an introduction to modeling and simulation using the Python programming language. The programs that we will be writing are very similar to their counterparts in several other languages, so our choice of language is not crucial. We will use Python because it is widely available, embraces a full set of modern abstractions, and has a variety of automatic checks for mistakes in programs, so it is suitable for learning to program. More importantly, Python is a free, open-source, high-level and interpreted general purpose programming language. Its simple syntax makes it suitable for learning programming as a first language. Other reasons for using Python:

- Python has a large community.
- Python has a very extensive standard library and thousands of external libraries.
- Python interoperates with other languages such as Fortran, C, C++, Java, etc.
- Python finds applications in areas such as:
 - Web Programming
 - Machine Learning and Artificial Intelligence
 - Data Science
 - Scripting and Automation

1.2 Your first program

As we dive into the world of Python programming we will first go through the basic steps required to get a simple program running. To get started we will need the Python **interpreter**. The Python **interpreter** (or simply Python) is a collection of applications, not unlike many of the other applications that you are accustomed to using (such as your word processor, email program, and web browser) that takes Python commands and translates them into a language that your operating system understands. As with any application, you need to be sure that Python is properly installed on your computer. You also need a text editor and a terminal application. Hopefully by now you

should have installed the Python programming environment using the Anaconda distribution. To test this, from the terminal type (on Linux/Darwin systems):

```
1 which python
```

and you will see something like:

```
1 /usr/bin/python
```

Once the Python interpreter is properly installed we can create our first program.

1.2.1 What is a Python program?

A Python program is nothing more than a sequence of characters stored in a file whose name has a *.py* extension. Python executes this sequence of characters in a specific, consistent, and predictable order - from top to bottom. Of course, just like any language - we need some rules, or syntax, and a text editor.

1.2.2 Hello, world!

The program *hello.py*, shown below, is an example of a complete Python program. The line numbers are shown to make it easy to reference specific lines, but they are not part of the program and should not be in your *hello.py* file.

```
1 print("Hello World!")
```

The program's sole action is to write a message back to the terminal window. The command `print()`, which writes the command to the screen, is an example of a Python **function**. We will discuss functions in great detail later.

1.2.3 Executing a Python Program

Once you compose the program, you can run it. When you run your program the Python **compiler** translates your program into a language that is more suitable for execution on a computer¹. Then the Python **interpreter** directs your computer to follow the instructions expressed in that language. Note that the **interpreter** is a loop² that:

1. Reads an expression
2. Evaluates the expression
3. Prints the result

If the result is **None**, the interpreter does not print it.

¹Though Python is known as an interpreted language, when you run a Python **program** the source code is compiled into a much simpler form called **bytecode**. This also happens at the Python interactive prompt. However, you will never notice this compilation steps because it is implicit.

²An interpreter is also called a "read-eval-print loop", or a REPL

1.2.4 Four Ways to Run Python

We will discuss 4 ways to run a Python program:

1. Interactive command line using **python**
2. Interactive command line using **ipython**
3. Scripting
4. Jupyter Notebook

Interactive Command Line

You can use the Python interpreter in an **interactive mode** to test some commands. For that you need to type **python** or **ipython** on the command line. Interactive code execution is recommended when you need to immediately test some fragment of the code. You can exit the interactive mode with `quit()` or `exit()` command or `Ctrl + D`. Note that the sequence of commands that you enter will not be saved if you close the interactive mode session.

What follows is a sample session using python

1. Open your terminal/Anaconda prompt and type:

```
python
```

This opens a Python **shell**, where you type statements one at a time.

2. Now type:

```
>>> print("Hello world!")
```

The `>>>` symbol is called a **prompt**.

3. You should see the output on the screen.
4. To exit the Python shell type `quit()` or `exit()` and the "return" key or simply `Ctrl + D`.

Using `ipython` is very similar. `ipython` is an enhanced interactive shell that has many features (e.g. tab-completion). It also has a prompt that is numbered.

1. Open your terminal/Anaconda prompt and type:

```
ipython
```

The prompt should look like the following:

```
In [1]:
```

2. Now type:

```
In [1]: print("Hello world!")
```

and press `Enter`.

3. You should see the output on the screen.
4. To exit the Python shell type `quit()` or `exit()` and the "return" key or simply `Ctrl + D`.

Scripting

Real Python programs are made as scripts and look like simple text files. These files generally have the extension `.py`. You can create text files, and thus Python scripts, using whatever text editor you like. To run a script you just need to use the Python interpreter and specify the name of the created file as an additional parameter.

```
1 $ python my_python_file.py
```

Sample session:

1. Create a new ASCII/text document named `hello.py` and enter the following text:

```
print('Hello world!')
```
2. Save the document and then in your terminal/Anaconda prompt, go to the directory containing the newly created file. From the terminal/Anaconda prompt, type:

```
$ python hello.py
```
3. You should see the output on the screen.

Python interpreter vs. Python program

Running a Python file as a program gives different results from pasting it line-by-line into the interpreter. In general the interpreter prints more output than the program would. That's because in the Python interpreter, evaluating a top-level expression prints its value while in a Python program, evaluating an expression generally does not print any output.

The Python Anaconda distribution comes with a very powerful integrated development environment (IDE) called [Spyder](#) that combines, among other things, an editor to compose programs and interactive tools for rapid testing.

Jupyter Notebook

A useful hybrid of the interactive terminal and the self-contained script is the **Jupyter Notebook**, a document format that allows executable code, formatted text, graphics, and even interactive features to be combined into a single document. Though the notebook began as a Python-only format, it has since been made compatible with a large number of programming languages. The notebook is useful both as a development environment, and as a means of sharing work via rich computational and data-driven narratives that mix together code, figures, data, and text.

What follows is a sample session using jupyter notebook

1. Open your terminal/Anaconda Prompt and type:

```
$ jupyter notebook
```
2. This directs you to a web browser and you can navigate to an already existing notebook or create one (right side menu New → Python 3). This will open up a new Untitled notebook where you can directly input Python code, Markup formatted text, or have raw text.

3. Now type:


```
print('Hello world!')
```
4. Press Shift+Enter, Cntrl+Enter or click Cells → Run Cells or use the Play button near the top of the page.
5. You should see the output on the screen. Note that both the prompt and the output look similar to what we saw with ipython.
6. Exit via closing the browser windows and stopping the server running in the terminal/command prompt (most likely with a Cntrl+C).

1.2.5 What to use?

For this class the recommendation will be to use the **Spyder** IDE. Spyder combines the editing and interactive functionalities integrated in one application. Python programs, which are just text files, are also easier to update and maintain.

1.3 Input and Output

So far we used the `print()` function to output data, the string "Hello, world!", to the standard output device (screen)³. To allow flexibility we might want to take the input from the user and provide the input to our programs to be processed and to eventually produce a result. There are two ways we can do that. First we have the `input()` function that allows user interactivity:

```
1 message = input('Enter a message: ')
2 print(message)
```

If you run this program it will wait for the user to enter some input. After entering the input one must press enter and then the next line containing the print statement is executed. We will rarely, if at all, use `input()`. More commonly, one uses command line arguments.

One example is illustrated in *useargs.py*. Open up the script in Spyder and look at its contents. If you run this program in Spyder you will get an error message because it is expecting a **command-line argument**. A **command-line argument** is text that you type after the program name on the **command line**. The program *useargs.py* takes one *argument* and writes it back out to the terminal as part of the message. For example, in your terminal run:

```
1 $ python useargs.py Alice
2 Hi, Alice. How are you?
3 $ python useargs.py Bob
4 Hi, Bob. How are you?
```

In *useargs.py*, the statement `import sys` tells Python that you wish to use the features defined in the *sys* **module**. One of those features, named *argv*, is a **list** of command-line arguments (which appear after `python useargs.py` on the command line, delimited by spaces). We will describe lists in detail; for now it is sufficient to understand that `sys.argv[1]` is the first command-line argument that you type after the program name, `sys.argv[2]` is the second command-line argument that you type after the program name and so forth.

³We can also output data to a file, but that subject will be discussed later.

1.4 Modules

As programs get bigger it is a good idea to break them up into modules. **Modules** are files containing Python definitions and statements. Python modules, such as **sys**, have a filename and end with the extension **.py**. Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import **keyword** to do this.

Which Python modules, besides *sys*, and functions are available to use? Many standard modules are bundled with any Python installation. Many others are available as extension modules that you can download and install subsequently. We can also compose our own. In short, thousands of Python modules, each (typically) defining multiple functions, are available for you to use. This manuscript introduces only the most fundamental modules and functions, and does so in a deliberately incremental fashion (starting in the next section) to avoid overwhelming you with information.

1.5 Intrinsic Functions

The **print()** function is a Python **built-in** function. Note that the general form of a function call is:

```
1 function_name(arguments)
```

The rules for executing a function call:

1. Evaluate the arguments.
2. Call the function, passing in the argument values.

Terminology:

1. Argument: a value given to a function
2. Pass: to provide to a function
3. Call: ask Python to evaluate a function
4. Return: pass back a value

Python has a set of built-in functions. To see the list of built-in functions, run

```
1 dir(__builtins__)
```

To get information about a particular function, call the **help** function and pass the function as the argument. For example:

```
1 help(print)
```

```

1 Help on built-in function print in module builtins:
2
3 print(...)
4     print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=
      False)
5
6     Prints the values to a stream, or to sys.stdout by default.
7     Optional keyword arguments:
8     file: a file-like object (stream); defaults to the current sys.
          stdout.
9     sep:  string inserted between values, default a space.
10    end:  string appended after the last value, default a newline.
11    flush: whether to forcibly flush the stream.

```

A useful definition to remember is:

An **argument** is a value you pass to a function when calling it.

1.5.1 Optional Arguments

If you run `help(pow)`:

```

1 help(pow)
2
3 you get
4
5 Help on built-in function pow in module builtins:
6
7 pow(x, y, z=None, /)
8     Equivalent to x**y (with two arguments) or x**y % z (with three
9     arguments)
10
11    Some types, such as ints, are able to use a more efficient
12    algorithm when
13    invoked using the three argument form.

```

In the documentation of function `pow`, the `z=None` indicates that the third argument `z` has a default value (which is `None`) and so `z` is optional. That is, function `pow` can be called with either two or three arguments:

```

1 pow(10, 2)
2 >>> 100
3 pow(10, 2, 3)
4 >>> 1

```

Can you figure out what the last statement is computing?

1.6 Errors

It is easy to blur the distinction among editing, compiling, and interpreting programs. You should keep them separate in your mind when you are learning to program, to better understand the effects of the errors that inevitably arise.

You can fix or avoid most errors by **carefully** examining the program as you create it. Some errors, known as *compile-time errors*, are raised when Python compiles the program, because they prevent the compiler from doing the translation. Python reports a compile-time error as a *SyntaxError*. Other errors, known as *run-time errors*, are not raised until Python interprets the program.

The most common problem among beginners involves misuse of Python's rules regarding whitespace characters such as tabs, spaces, and newlines. In general, Python considers most whitespace in program text to be equivalent, with two important exceptions: string literals and indentation. A **string literal** is a sequence of characters inside single quotes, such as "Hello, World". If you put any number of spaces within the quotes, you get precisely the same number of spaces in the string literal. **Indentation** is whitespace at the beginning of a line. The number of spaces at the beginning of a line plays an important role in structuring Python programs, as we will see later in the course. For now you should not indent any code.

1.7 Programming Style

One final item that deserves some elaboration is programming style.

The overarching goal when composing code is to make it easy to understand. Understandable programs are more likely to be correct, and are more likely to stay correct as they are maintained over time. One simple way to write good programs is to use comments. Comments are indispensable because they help other programmers to understand your code and even can help you to understand your own code in retrospect. Whereas constraints in the course, and this manuscript, demand that we use comments sparingly *you* should strive to write good comments.

Programmers use style guides to make programs easier to understand. The official Python style guide is given in <http://www.python.org/dev/peps/pep-0008/>. We recommend that you give the style guide a quick read now, and that you return to it occasionally as you gain more experience with composing Python programs.

1.8 References

- You are encouraged to visit the official Python website, <http://www.python.org>. More specifically:
[Official Documentation \(Python 3.x\)](#)
- The Python Visualizer (written by Philip Guo) is a great tool that helps visualize program execution:
[Python Tutor](#)

Chapter 2

Variables and Data Types

2.1 Fundamentals

2.1.1 Operators

An operator is a symbol that represents an operation that may be performed on one or more operands. For example, the + symbol represents the operation of addition. An operand is a value that a given operator is applied to, such as operands 2 and 3 in the expression 2 + 3.

An **operator** is a symbol that represents an operation that may be performed on one or more operands. Operators that take one operand are called **unary** operators. Operators that take two operands are called **binary** operators.

2.1.2 Summary of Python Arithmetic Operators

+ Addition	Adds values on either side of the operator.
- Subtraction	Subtracts right hand operand from left hand operand.
* Multiplication	Multiplies values on either side of the operator
/ Division	Divides left hand operand by right hand operand
// Floor Division	Returns integer part of the quotient
% modulo	Divides left hand operand by right hand operand and returns remainder
** Exponent	Performs exponential (power) calculation on operators

Python provides two forms of division. "true" division is denoted by a single slash, /. Thus, 25/10 evaluates to 2.5. Truncating division is denoted by a double slash, //, providing a truncated result based on the type of operands applied to.

2.1.3 Python Expressions

An **expression** is a combination of symbols (or single symbol) that evaluates to a value. Expressions, most commonly, consist of a combination of operators and

```
operands.
```

Open up the Python interpreter and type the following **expressions**:

```
1 2
2 1 + 2
3 2**12
4 1/-12
5 (72 - 32)/9*5
```

Python will happily compute their values. The first three expressions are straightforward. The fourth one would be considered very unusual or even confusing if handwritten on a piece of paper but in Python it is unambiguously correct. What about the last one? In Python an expression is evaluated from the **inside out**¹. So, the expression $(72 - 32)/9 * 5$ is evaluated as follows:

```
1 (72 - 32)/9*5
2 (40)/9*5
3 40/9*5
4 4.44*5
5 22.2
```

Though this may seem trivial note what happens when you enter the following expression $(72 - 32)/(9 * 5)$? What do you get? 0.88. Well, perhaps that's what you want to compute. However, if you are trying to convert degrees Fahrenheit to degrees Celsius then the last expression (and result) is wrong. So, **precedence of operators is important in Python** and if precedence is not clear then you should use **parentheses**².

When Python executes the following expressions there are differences between integer arithmetic and real (floats) arithmetic that you should keep in mind (You can do this just in your interpreter and you don't need to turn anything in for this part, but pay attention to the output!)

```
1 5/2
2 5/2.0
3 5.0/2
4 7*(1/2)
5 7*(1/2.0)
6 5**2
7 5.0**2
8 5**2.0
9 1/3.0
```

Note that as long as one argument is a **float** all results will be floats. In the last case the final digit is rounded. Python does this for non-terminating decimal numbers, as computers cannot store infinite numbers!

¹More generally, Python evaluates an expression by first evaluating its sub-expressions, then performing an operation on the value. Notice that each sub-expression might have its own sub-sub-expressions, so this process might repeat several times.

²If you remember **PEMDAS** from elementary school then it is the same for Python: (),**,*/,+,-

2.2 Keywords and Identifiers

Keywords are the **reserved words** in Python. Keywords cannot be used as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language. In Python, keywords are case sensitive and all keywords - except True, False and None - are in lowercase.

The list of Python keywords (as of Python version 3.7) is:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Table 2.1: Keywords in Python

An **identifier** is a name given to entities like variables, functions and **classes** to help them differentiate one from another.

2.3 Variables

Think of a variable as a container. A variable stores a value so that you can reuse it later in your program. This reduces redundancy, improves performance, and makes your code more readable. In order to use a variable, you first store a value in the variable by assigning the variable to this value. Later, you access that variable, which looks up the value you assigned to it. It is an error to access a variable that has not yet been assigned. You can reassign a variable - that is, give it a new value - any number of times.

Note that Python's concept of a variable is different from the mathematical concept of a variable. In math, a variable's value is fixed and determined by a mathematical relation. In Python, a variable is assigned a specific value at a specific point in time, and it can be reassigned to a different value later during a program's execution.

Python stores variables and their values in a structure called a **frame**. A frame contains a set of **bindings**. A binding is a relationship between a variable and its value. When a program assigns a variable, Python adds a binding for that variable to the frame (or updates its value if the variable already exists). When a program accesses a variable, Python uses the frame to find a binding for that variable.

2.3.1 Assignment statements

An assignment statement is a directive to Python to **bind** the variable on the left side of the = operator to the object produced by evaluating the expression on the right side. For example, when we write `c = a + b`, we are expressing this action: "associate the variable c with the sum of the values associated with the variables a and b."

In lecture we discussed how one can assign values to a variable. Let's look at that in more detail. Consider the following series of statements where we introduce the Python built-in function `id` that returns the memory address used by the variable:

```
1 In [1]: x = 2
2 In [2]: print(id(x), x)
3 4490380384 2
```

That big number 4490380384 denotes where the data lives in the memory and it will probably be different in your computer system. What happens if we create another variable with the same value?

```
1 In [3]: y = 2
2 In [4]: print(id(y), y)
3 4490380384 2
```

After two consecutive assignments the *ids* of both *x* and *y* are the same implying that we are reusing the same memory location. Python does this to *optimize* memory and only so for very special cases (in the above case for **small** integers)!

For now, the take home message is that "=" in an assignment statement is different than the mathematical meaning of "=". Evaluating an expression gives a new (copy of a) number, rather than changing an existing one. A useful tool to visualize how an assignment statement is executed is the [Python Visualizer](#). Try it!

2.3.2 Namespaces

A **name** (also called identifier) is simply a name given to objects. Everything in Python is an object. A name is a way to access the underlying object.

So now that we understand what names are, we can move on to the concept of **namespaces**: A **namespace** is a collection of names. Imagine a namespace as a mapping of every name, you have defined, to corresponding objects. Different namespaces can co-exist at a given time but are completely isolated.

A namespace containing all the built-in names is created when we start the Python interpreter and exists as long we don't exit. This is the reason that built-in functions like `id()`, `print()` etc. are always available to us from any part of the program. To view the global namespace run:

```
1 dir()
```

Each module creates its own global namespace. These different namespaces are isolated. Hence, the same name that may exist in different modules do not collide. Modules can have various functions and classes (which we have not discussed yet). This leads to the idea of variable scope which will discuss later.

2.4 Built-in Data Types

2.4.1 Fundamental Types

A data type is a set of values and a set of operations defined on those values.
--

Many data types are built into the Python language. So far, each value we have seen is a single datum, such as an integer, decimal number, or Boolean. We will now formally introduce Python's built-in data types **int** (for integers), **float** (for floating-point numbers), **str** (for sequences of characters) and **bool** for logical variables. But first, we introduce an important concept: **objects**.

Objects

All data values in a Python program are represented by **objects** and relationships among objects. An object is an in-computer-memory representation of a value from a particular data type. Each object is characterized by its **identity**, **type**, and **value**.

- The identity uniquely identifies an object. You should think of it as the location in the computer's memory (or memory address) where the object is stored.
- The type of an object completely specifies its behavior - the set of values it might represent and the set of operations that can be performed on it.
- The value of an object is the data-type value that it represents.

Each object stores one value; for example, an object of type `int` can store the value 1234 or the value 99 or the value 1333. Different objects may store the same value. For example, one object of type `str` might store the value 'hello', and another object of type `str` also might store the same value 'hello'. We can apply to an object any of the operations defined by its type (and only those operations). For example, we can multiply two `int` objects but not two `str` objects.

Integers

The **int** data type represents integers or natural numbers. The common arithmetic operations on integers have already been introduced.

Floats

The **float** data type is for representing floating-point numbers, for use in scientific and commercial applications. The common arithmetic operations for integers also work with floats. A very useful built-in function in Python is the **type()** function that prints the type of the variable:

```
1 i = 2
2 print(type(i))
3 Out []:
4 <class 'int'>
5 x = 3.14
6 print(type(x))
7 Out []:
8 <class 'float'>
```

We use floating-point numbers to represent real numbers, but they are decidedly not the same as real numbers! There are infinitely many real numbers, but we can represent only a finite number of

floating-point numbers in any digital computer. For example, $5.0/2.0$ evaluates to 2.5 but $5.0/3.0$ evaluates to 1.6666666666666667. Typically, floating-point numbers in Python are represented in **double precision** and have 15-17 decimal digits of precision.

Strings

The **str** data type represents strings, for use in text processing. The value of a str object is a sequence of characters. You can specify a str literal by enclosing a sequence of characters in matching single quotes or double quotes.

```
1 str1 = 'this is a string'
2 str2 = "this is also a string"
```

Strings can not be changed, i.e. they are **immutable**³. This simply means that once a string is created it maintains its **identity** forever. If it is redefined then its identity is changed. For example:

```
1 str1 = 'this is a string'
2 str1 = 'this is also a string'
```

In this case the second instance of str1 has the same name as the first instance, but a different identity.

One can create a new string from the concatenation of two strings using the operator +.

```
1 str1 = "abcde"
2 str2 = "efghi"
3 str3 = str1 + str2
```

Creating a string of repeating characters is accomplished by using the multiplication sign.

```
1 str1 = "*"
2 10*str1
3
4 Out []: '*****'
```

Converting numbers to strings for output. Python provides the built-in function **str()** to convert numbers to strings. Our most frequent use of the string concatenation operator is to chain together the results of a computation for output using the print function, often in conjunction with the str() function, as in this example:

```
1 x = 1
2 y = 2
3 print(str(x) + '+' + str(y))
```

Converting strings to numbers for input. Python also provides built-in functions to convert strings (such as the ones we type as command-line arguments) to numeric objects. We use the Python built-in functions **int()** and **float()** for this purpose. If the user types 1234 as the first command-line argument, then the code **int(sys.argv[1])** evaluates to the int object whose value is 1234.

³**int** and **float** data types are also immutable. That is, once you create a numeric variable, you cannot change it. It makes sense, right?

Accessing String Characters

A string is a sequence of characters. For example:

```
1 a_str_seq = "We bring together a multitude of people and ideas in
                everything that we do. Our culture of inclusion, our
2 multidisciplinary approach, and our global perspective, make us more
                effective educators and scholars."
3 a_str_seq[0]
4
5 Out []: 'W'
6 a_str_seq[1]
7
8 Out []: 'e'
```

The value in the square brackets is the **index**. Python, like C and Java, begins counting at 0, and so the first position in a string is denoted by a 0, the second position in a string is denoted by a 1, and so on.

Negative indices will access letters from the end of the string. So,

```
1 a_str_seq[-1]
2
3 Out []: '.'
4 a_str_seq[-2]
5
6 Out []: 's'
```

shows two examples which retrieve the last character (using [-1]) and the next to last character (using [-2]).

Accessing multiple letters of a string is performed through **slicing**. For example:

```
1 a_str_seq[0:6]
2 Out []: 'We bri'
```

The first is the starting index and the second is the ending index. However, the character of the second index is not extracted. Therefore `a_str_seq[0:6]` extracts letters from positions 0, 1, 2, 3, 4 and 5 but not 6. The first index is included and the second index is excluded.

The example below shows the task of splitting a string up into two substrings at location 10. Line 1 uses `[:10]` instead of `[0:10]`. These are equivalent. If the substring being extracted starts at the beginning of the string, then the first index is not required. Likewise, if the extraction goes to the end of the string, then the second index is not required as shown in line 2. The string `a1` is the first ten characters of the original string, and `a2` is the rest of the string. Since the second index of line 1 is excluded and the first index of line 2 is included, the character at position 10 appears in only one of the strings.

```
1 a1 = a_str_seq[:10]
2 a2 = a_str_seq[10:]
```

In the next example 3 values are used. These are the starting index, the ending index, and the step size. In this example, the a3 is every third character from the first 20 characters of a_str_seq.

```
1 a3 = a_str_seq[0:20:3]
2 a3
3
4 Out []: 'Wbnteea'
```

Finally, the next example has nothing before the first or second colon. This indicates that the entire string is being used. The step size is -1 which steps in the backwards direction. Thus, this line reverses the string. The first 10 characters of this new string are printed to confirm the result.

```
1 a4 = a_str_seq[::-1]
2 a4
3
4 Out []: '.sralohcs '
```

String Functions

Several functions are defined to manipulate strings and return information about their contents. Examples:

- The **len** function determines the length (numeric size) a a string.
- The **find** function will find the location of the first substring within a string.
- The **count** function counts the number of occurrences of a target string.
- The case of a string can be forced by the functions **upper** and **lower**.
- The **replace** function replaces a substring with another.

Usage:

```
1 In [1]: my_str = "Don't judge each day by the harvest you reap but
           by the seeds that you plant."
2
3 In [2]: len(my_str)
4 Out [2]: 77
5
6 In [3]: my_str.find('e')
7 Out [3]: 10
8
9 In [4]: my_str.find('y')
10 Out [4]: 19
11
12 In [5]: my_str.count("you")
13 Out [5]: 2
14
15 In [6]: my_str.upper()
```

```

16 Out [6]: "DON'T JUDGE EACH DAY BY THE HARVEST YOU REAP BUT BY THE
           SEEDS THAT YOU PLANT."
17
18 In [7]: my_str.lower()
19 Out [7]: "don't judge each day by the harvest you reap but by the
           seeds that you plant."
20
21 In [8]: my_str.replace('y', 'Y')
22 Out [8]: "Don't judge each daY bY the harvest You reap but bY the
           seeds that You plant."

```

Booleans

The bool data type has just two values: True and False. The apparent simplicity is deceiving because booleans lie at the foundation of computer science. The most important operators defined for booleans are the logical operators: *and*, *or*, and *not*.

isinstance

We can use the **isinstance** built-in function for testing types of variables:

```

1 isinstance(x, float)
2 True

```

Finally, you can do **type casting**:

```

1 x = 1.5
2 print(x, type(x))
3 (1.5, <type 'float'>)
4 x = int(x)
5 print(x, type(x))
6 (1, <type 'int'>)

```

2.5 Formatting Text and Numbers

From Newton's second law of motion one can set up a mathematical model for the motion of the ball and find that the vertical position of the ball, called y , varies with time t according to the following formula:

$$y(t) = v_0t + \frac{1}{2}gt^2 \quad (2.1)$$

Instead of just printing the numerical value of y in our programs, we may want to write a more informative text, typically something like

```

1 at t= 0.6 s, the height of the ball is 1.23 m.

```

where we also have control of the number of digits (here y is accurate up to centimeters only). How can we do that? One way to do it is to use Python's `str.format()`. `format()` is a function available to string objects that provides the ability to do complex variable substitutions and value formatting.

The built-in `format` function can be used to produce a numeric string of a given floating-point value rounded to a specific number of decimal places.

2.5.1 Number Formatting

The following table shows various ways to format numbers⁴ using Python's `str.format()`, including examples for both float formatting and integer formatting.

To run examples use `print("FORMAT".format(NUMBER))`. So, to get the output of the first example, you would run:

```
1 print("{:.2f}".format(3.1415926));
```

Number	Format	Output	Description
3.1415926	<code>{:.2f}</code>	3.14	2 decimal places
2.71828	<code>{:.0f}</code>	3	No decimal places
-1	<code>{:+.2f}</code>	-1.00	2 decimal places with sign
0.25	<code>{:.2%}</code>	25.00%	Format percentage
1000000000	<code>{:.2e}</code>	1.00e+09	Exponent notation
5	<code>{:0>2d}</code>	05	Pad integer with zeros (left padding, width 2)

2.5.2 string.format() basics

Here are a couple of examples of basic string substitution, the `{}` is the placeholder for substituted variables. If no format is specified, it will insert and format as a string.

```
1 s1 = "Python is {}".format("a very popular language")
2 s2 = "CDS230 combines {} and {} elements".format("data", "science")
```

You can also use the numeric position of the variables and change them in the strings, this gives some flexibility when doing the formatting, if you make a mistake in the order you can easily correct without shuffling all the variables around.

```
1 s1 = " {0} is better than {1} ".format("emacs", "vim")
2 s2 = " {1} is better than {0} ".format("emacs", "vim")
```

Now we can format the output at the beginning of this section:

```
1 t = 0.6
2 y = 1.23456
3 print("at t= {} s, the height of the ball is {:.2f} m.".format(t,y))
```

⁴There are many more ways. These are the ones we'll use in this class. For more information see the Python documentation.

2.5.3 F-strings

The same formatting can be accomplished using f-strings. With f-strings we do away with the `.format()` syntax and instead prepend the string with an "f" and use string substitution as before. For example:

```
1 t = 0.6
2 y = 1.23456
3 print(f"at t= {t} s, the height of the ball is {y:.2f} m.")
```

For this class you can use either `format()` or f-strings

2.6 Problem Solving with Python

The solution to most of the exercises in this course is a Python program. To produce the solution, you first need understand the problem and what the program is supposed to do, and then you need to understand how to translate the problem description into a series of Python statements (see Appendix B). Equally important is the verification (testing) of the program. A complete solution to a programming exercise therefore consists of two parts: the program text and a demonstration that the program works correctly. Some simple programs have so simple output that the verification can just be to run the program and check the output. In cases where the correctness of the output is not obvious, it is necessary to convince yourself that the result is correct. How? This can be a calculation done separately on a calculator or one can apply the program to a special simple test with known results. We will deal with these issues as they arise.

What follows is a series of examples that aim to show how to go about solving problems with Python in this course.

Example 1: Write a program for converting Fahrenheit degrees to Celsius.

Solution: The solution process can be divided into three steps:

1. Establish the mathematics to be implemented. Do we know the conversion formula? Yes, the formula to use is $C = \frac{5}{9}(F - 32)$, where F is the input temperature in degrees Fahrenheit and C is the desired output in Celsius.
2. Coding of the formula in Python. This can be as simple as $C = (5/9) * (F - 32)$.
3. Establish some test cases. This is very important. The important idea is, if possible, to use test cases that we know the answer to, generally boundary cases. For example we all know that $F = 32$ corresponds to $C = 0$. We can therefore, in our new program, set $F = 32$ and check that we get $C = 0$.

```
1 # Convert from Fahrenheit degrees to Celsius degrees
2 F = 32
3 C = (5/9)*(F - 32)
4 print(C)
5 Out []: 0.0
```


Another test case could be $F = 212$ corresponding to $C = 100$, the boiling point of water. Once you are satisfied that your program passes some simple tests you can gain confidence that it *may* be correct. The current problem is simple enough to test but that will very frequently not be the case.

Example 2 (trig_identity.py): Show that $\sin^2\theta + \cos^2\theta = 1$.

Solution: This is not expected to be a mathematical proof but rather a computational one. One way of doing this is as follows:

```
1 from math import sin, cos, pi # import only what you need
2 theta = pi/4 # test value in radians
3 one = sin(theta)**2 + cos(theta)**2
4 print(one)
5 Out []: 1.0
```

Remember: when dealing with trig functions all angles are represented in radians. Here we only tested one value. Is that enough?

Example 3 (quad_math_error.py): Very often we find ourselves trying to figure out why our program doesn't work. So, can you find the problem(s) with the following program?

```
1 from math import sqrt # import only sqrt
2 a = 2
3 b = 1
4 c = 2
5 q = sqrt(b*b - 4*a*c)
6 x1 = (-b + q)/2*a
7 x2 = (-b - q)/2*a
8 print(x1, x2)
```

Upon running the program we will get the following output:

```
1 Traceback (most recent call last):
2   File "quad_math_error.py", line 3, in <module>
3     q = sqrt(b*b - 4*a*c)
4 ValueError: math domain error
```

The Python interpreter will point you where the error is occurring and the error message says that the value is wrong. You can probably check manually and note that the value inside the square root is negative. Therefore, to fix the problem you need to be able to deal with negative roots, that is, use complex numbers. *Complex* numbers and functions can be imported using the **cmath** module. So, changing "from math import sqrt" to "from cmath import sqrt" will fix the problem.

Example 4 (trajectory.py): **Trajectory of a ball.** One can show that a ball thrown at an angle θ with the horizontal ball will follow a trajectory through the air described by the following equation:

$$f(x) = x \tan\theta - \frac{1}{2v_0^2} \frac{gx^2}{\cos^2\theta} + y_0 \quad (2.2)$$

In this expression, x is a horizontal coordinate, g is the acceleration of gravity, v_0 is the magnitude of the initial velocity which makes an angle θ with the x axis, and y_0 is the initial position of the

ball. Suppose a baseball is hit at 60 km/h at an initial angle of 30° . Also, let $x = 0.5\text{m}$ and $y_0 = 1\text{m}$. Write a program for evaluating $f(x)$. The program should write out the value of all the involved variables and what their units are.

Solution: We use the **SI** system and so v_0 is given in m/s ; x , y , and y_0 are measured in meters; θ in radians and $g = 9.81 \text{ m/s}^2$. The program has naturally four parts: initialization of input data, import of functions and π from `math`, conversion of v_0 and θ to m/s and radians, respectively, and evaluation of $f(x)$. We choose to write out all numerical values with one decimal. The program could look like this:

```

1
2 # Initialization
3 g = 9.81      # m/s**2
4 v0 = 60      # km/h
5 theta = 30   # degrees
6 x = 0.5      # m
7 y0 = 1       # m
8
9 # Import functions
10 from math import pi, tan, cos
11
12 # print out all variables
13 print("""\
14 v0      = {:.1f} km/h
15 theta   = {:d} degrees
16 y0      = {:.1f} m
17 x       = {:.1f} m\
18 """).format (v0, theta, y0, x)
19 )
20
21 # Convert v0 to m/s and theta to radians
22 v0 = v0/3.6
23 theta = theta*pi/180
24
25 # Evaluate f(x)
26 y = x*tan(theta) - 1/(2*v0**2)*g*x**2/((cos(theta))**2) + y0
27
28 # Solution to problem
29 print('y      = {:.1f} m' .format(y))
30
31 Output:
32
33 v0      = 60.0 km/h
34 theta   = 30 degrees
35 y0      = 1.0 m
36 x       = 0.5 m
37 y       = 1.3 m

```

Note how we use a `print()` with a format statement spanning multiple lines (line 12).

Example 5 (age_secs.py): Age in Seconds Program

We look at the problem of calculating an individual's age in seconds. It is not feasible to determine a given person's age to the exact second. This would require knowing, to the second, when they were born. It would also involve knowing the time zone they were born in, issues of daylight savings time, consideration of leap years, and so forth. Therefore, the problem is to determine an *approximation* of age in seconds. The program will be tested against calculations of age from online resources.

So, how do we get started? This is a complex problem so we will follow the guidance from appendix B.

The Problem Statement

Determine the approximate age of an individual in seconds, say, within 99% accuracy of results from online resources. The program must work for dates of birth from January 1, 1900 to the present.

Problem Analysis

The fundamental computational issue for this problem is the development of an algorithm incorporating approximations for information that is impractical to utilize (time of birth to the second, daylight savings time, etc.), while producing a result that meets the required degree of accuracy.

Program Design

There is no requirement for the form in which the date of birth is to be entered. We will therefore design the program to input the date of birth as integer values. Also, the program will not perform input error checking, since we have not yet covered the programming concepts for this.

Data Description

The program needs to represent two dates, the user's date of birth, and the current date and then calculate the difference. Since each part of the date must be able to be operated on arithmetically, dates will be represented by three integers. For example, May 15, 1992 would be represented as follows:

```
1 year=1992
2 month=5
3 day=15
```

Though these are integers, we cannot just take the difference of two dates. How can we deal with dates? That can be a bit difficult and can lead us down a rabbit hole. So, we will use the Python Standard Library module **datetime** to represent dates and which contains lots of functionality to help us deal with datetime arithmetic (see <https://docs.python.org/3.7/library/datetime.html>).

Now we consider how the calculations can be approximated without greatly affecting the accuracy of the results. First we start with the issue of leap years. Since there is a leap year once every four

years (with some exceptions), we calculate the average number of seconds in a year over a four-year period that includes a leap year. Since non-leap years have 365 days, and leap years have 366, we need to compute,

```

1 numsecs_day = (hours per day) * (mins per hour) * (secs per minute)
2 numsecs_year = (days per year) * numsecs_day
3 avg_numsecs_year = ((4 * numsecs_year) + numsecs_day) // 4
4 avg_numsecs_month = avg_numsecs_year // 12

```

Note that if we directly determined the number of seconds between the date of birth and current date, the months and days of each would need to be compared to see how many full months and years there were between the two. Using 1900 as a basis avoids these comparisons. Thus, the rest of our algorithm is given below.

```

1 numsecs_1900_to_dob = (year_birth - 1900) * avg_numsecs_year +
2     (month_birth - 1) * avg_numsecs_month +
3     (day_birth * numsecs_day)
4 numsecs_1900_to_today = (current_year - 1900) * avg_numsecs_year +
5     (current_month - 1) * avg_numsecs_month +
6     (current_day * numsecs_day)
7 age_in_secs = num_secs_1900_to_today - numsecs_1900_to_dob

```

Program Implementation and Testing First, we decide on the variables needed for the program. For date of birth, we use variables month_birth, day_birth, and year_birth. Note that this is better than using variables such as m, d and y. Similarly, for the current date we use variables current_month, current_day, and current_year.

```

1 import datetime
2
3 # Inputs
4 month_birth = int(input('Enter month born (1-12): '))
5 day_birth = int(input('Enter day born (1-31): '))
6 year_birth = int(input('Enter year born (4 digit): '))
7
8 # Get current time
9 current_month = datetime.date.today().month
10 current_day = datetime.date.today().day
11 current_year = datetime.date.today().year
12
13 # test output:
14 print(f"Your birth date is: {month_birth} {day_birth} {year_birth}")
15 print(f"The current date is: {current_month} {current_day} {
16     current_year}")
17
18 # Main algorithm
19 numsecs_day = 24*60*60
20 numsecs_year = 365*numsecs_day
21 avg_numsecs_year = ((4 * numsecs_year) + numsecs_day) // 4

```

```

22 avg_numsecs_month = avg_numsecs_year // 12
23
24 numsecs_1900_to_dob = (year_birth - 1900) * avg_numsecs_year + \
25     (month_birth - 1) * avg_numsecs_month + \
26     (day_birth * numsecs_day)
27 numsecs_1900_to_today = (current_year - 1900) * avg_numsecs_year + \
28     (current_month - 1) * avg_numsecs_month + \
29     (current_day * numsecs_day)
30 age_in_secs = numsecs_1900_to_today - numsecs_1900_to_dob
31
32 print(f"You are approximately {age_in_secs} seconds old")

```

Notes

- Note how `input()` is used: Each input is **type casted** to an integer.
- Note the use of integer division `//`
- Note the use of line continuation characters `\`

Run the program and compute your age in seconds. Compare the result with those of an online program. Do you think the program above is "good enough"?

Chapter 3

Control Flow

Control flow refers to the order that instructions are executed in a program.

A **control statement** is a statement that determines the control flow of a set of instructions.

There are three fundamental forms of control that programming languages provide - *sequential control*, *selection control*, and *iterative control*. Collectively a set of instructions and the control statements controlling their execution is called a **control structure**.

3.1 Boolean Expressions (Conditions)

Each value in Python has a type: int, float, string, boolean, etc. A boolean can have either the value True or the value False. In Python, certain operators compute values that are True or False.

An expression that computes a True or False value is called a **boolean expression**.

3.1.1 Conditional Operators

There are several conditional operators:

- < less than
- > greater than
- == equal to
- >= greater than or equal to
- <= less than or equal to
- != not equal to

These operators not only apply to numeric values, but to any set of values that has an ordering, such as strings. Examples:

```
1 print( True and True )      # prints True
2 print( True and False )    # prints False
3 print( 3 < 4 and 10 < 12 )  # prints True
4 print( 3 < 4 or 12 < 10 )   # prints True
5 print( 4 < 3 or 12 < 10 )   # prints False
6 print( (4 < 3 and 12 < 10) or 7 == 7 ) # prints True
7 print(10 < 0 and not 10 > 2) # prints False
8 'Alice' < 'Bob' # prints True
```

String values are ordered based on their character encoding, which normally follows a **lexographical (dictionary) ordering**. So in the last example, 'Alice' is less than 'Bob' because the Unicode (ASCII) value for 'A' is 65, and 'B' is 66.

When we have a boolean expression like $x < 4$ Python actually computes a value. In fact, it computes a boolean value of True or False. So if x currently has the value 5, the expression $x < 4$ evaluates to the value False. It follows that you can store the results of a conditional operation in a variable:

```
1 z = 1 > 2
2 print(z)
3 False
```

Notice that in mathematics, the equation $z = 1 > 2$ makes no sense. In Python, the line of code $z = 1 > 2$ is perfectly fine. It means: compute the expression $1 > 2$, which gives False, and then assign that False value into the variable z .

Consider the following example:

```
1 from math import pi, sin
2
3 print( pi )                # prints 3.14159265359
4 print( sin(pi) )          # prints 1.22464679915e-16
5 print( sin(pi) == 0 )     # prints False. Uh-oh!
```

The problem you see above arises because floats have limited precision. That is, Python only has an approximate value for π . The numerical computation of the *sin* function is also approximate. So, careful when using `==` to compare floats!. Can you think of a more robust way to compare floats?

3.1.2 Logical Operators

One can operate on boolean values using logical operators. *and*, *or* and *not* are Python's logical operators that operate on boolean values and evaluate to another boolean value. Interpretation of logical expressions involving not, or, and is straightforward when the operands are Boolean:

- *not*. Logically reverses the sense of x .
- *and*. Given x and y expression evaluates to True if both x and y are True, False otherwise.

- *or*. Given **x or y** expression evaluates to True if either x or y are True, False otherwise.

Caveat: Notice that if the first operand of *and* evaluates to False, we're done: we know that the result of *and* must be False, regardless of the second operand. We don't even have to look at the second operand. Python **short-circuits** if it sees that the first operand of an *and* is False; it doesn't evaluate the second operand at all!

In **short-circuit** evaluation, the second operand of Boolean operators **and** and **or** is not evaluated if the value of the Boolean expression can be determined from the first operand alone.

Finally, it is interesting to note that in Python **every object has a boolean value**. Generally one finds that:

- All integers evaluate to True, except 0 which evaluates to False
- All strings evaluate to True, except the empty string

One can use the boolean function **bool** to evaluate any Python object and check its boolean value. For example `bool(True)` returns True and `bool(1<2)` return False. Run the following examples on the interpreter and try to understand the output:

```
1 bool("hello")
2 bool(1 and 1)
3 bool(0 and "test")
4 bool(False or 1)
5 bool(True and 10 or not 0)
```

3.2 Conditionals

All of the programs that we have examined to this point have a simple flow of control: the statements are executed one after the other in the order given. Most programs have a more complicated structure where statements may or may not be executed depending on certain conditions (conditionals), or where groups of statements are executed multiple times (loops).

3.2.1 if Statements

In Python conditionals are known as control structures because they direct the order of execution of the statements in a program. There are various structures depending on how many conditions are being evaluated.

Unary Selection

```
1 if condition:                                # HEADER
2     Python code that runs iff condition is True # CLAUSE
3     Proper indentation is critical
```


This is the simplest control structure. There is one condition that, if True, evaluates the statement(s) in the clause else it does nothing. First of all note that there is a colon after the condition.

More important is the amount of **indentation** of each program line. In most programming languages, indentation has no affect on program logic - it is simply used to align program lines to aid readability. In Python, however, indentation is used to associate and group statements. In fact, all statements within a Python **block**¹ must same the same indentation². Example:

```
1 y = -2
2 a = y < 1
3 if a:
4     print ('a is non-zero')
```

Binary Selection: Binary selection has two conditions:

```
1 if condition:
2     Python code that runs iff condition is True
3 else:
4     Python code that runs iff condition is False
5     ...again, indentation is important
```

This control structure divides the flow in two depending on whether the control condition is True or False. In the following example **input** is a Python function that prompts a user for input. **input** always expects a **string** which is why *n* is converted to an *int* via **type casting**:

```
1 n = int(input('Enter a number: '))
2 if n % 2 == 0:
3     print ("Number is even")
4 else:
5     print ("Number is odd")
6 print ("Done")
```

If $n \% 2$ is true, the first clause is executed, and the second is skipped. If $n \% 2$ is false, the first clause is skipped and the second is executed. Either way, execution then resumes after the second clause. Both clauses are defined by indentation.

Chained if Statements: There is also syntax for branching execution based on several alternatives. For this, use one or more **elif** (short for else if) clauses. Python evaluates each expression in turn and executes the clause corresponding to the first that is true. If none of the expressions are true, and an else clause is specified, then its clause is executed³:

```
1 if condition1:
2     Python code that runs iff condition1 is True
3 elif condition2:
4     Python code that runs iff condition2 is True
5 elif condition3:
6     Python code that runs iff condition3 is True
7 else:
8     Python code that runs iff conditions 1-3 are False
```

¹The usual approach taken by most programming languages is to define a syntactic device that groups multiple statements into one compound statement or block. A block is regarded syntactically as a single entity.

²In Python, 4 spaces is the standard

³Note the else clause is optional.

Example:

```
1 n = int(input('Enter a number: '))
2 if n < 0:
3     print ('n is negative')
4 elif n > 0:
5     print ('n is positive')
6 else:
7     print ('n is zero')
```

An if statement with elif clauses uses short-circuit evaluation, analogous to what you saw with the and and or operators. Once one of the expressions is found to be true and its block is executed, none of the remaining expressions are tested.

Nested if Statements: This is a generalization of the above cases.

```
1 if condition1:
2     Python code that runs iff condition1 is True
3 else:
4     Python code that runs iff condition1 is False
5     if condition2:
6         Python code that runs iff condition2 is True
7     else:
8         Python code that runs iff condition2 is False
```

Example:

```
1 n = int(input('Enter a number: '))
2 if n > 0:
3     print ('n is positive')
4     if n % 2 == 0:
5         print ('...and also even')
6     else:
7         print ('...and also odd')
8 else:
9     if n == 0:
10        print ('n is zero')
11    else:
12        print ('n is negative')
```

Conditionals allow us to write programs that are more interesting than "straight-line" programs, but it is still quite limited, and often misused or abused.

For example:

```
1 # code that computes precipitation and temperature
2 #...
3 precipitation = True
4 temperature = 2.0
5 #...
6 #
7 if precipitation:
```

```

8     if temperature > 0:
9         print('Bring your umbrella!')
10    else:
11        print('Wear your snow boots and winter coat!')

```

The statement above can be simplified by removing some of the nesting. The message 'Bring your umbrella!' is printed only when both of the if statement conditions are True. The message 'Wear your snow boots and winter coat!' is printed only when the outer if condition is True, but the inner if condition is False. The following is equivalent to the code above:

```

1 precipitation = True
2 temperature = 2.0
3 if precipitation and temperature > 0:
4     print('Bring your umbrella!')
5 elif precipitation:
6     print('Wear your snow boots and winter coat!')

```

One way to think about the power of a class of programs is in terms of how long they can take to run. Assume one line of code takes one unit of time to execute. If a "straight-line" program has n lines of code, it will take n units of time to execute. What about a program with selection statements? It might take less than n units of time to run but it cannot take more since each line of code is executed at most once.

A program for which maximum running time is bounded by the length of the program is said to run **constant in time**. Constant-time programs are quite limited in what they can do. The study of intrinsic difficulty of problems is the topic of **computational complexity**. We may allude to this topic a few times later in the semester.

3.3 Loops

If there is one thing computers are good for is to perform repetitive tasks. For that reason loop constructs are some of the most useful ones in programming⁴ and there are two types: indefinite and definite.

A **definite** loop is a program loop in which the number of times the loop will iterate can be determined before the loop is executed. A **indefinite** loop is a program loop in which the number of times the loop will iterate is not known before the loop is executed.

3.3.1 Indefinite Iteration: while Loops

A while loop is similar to an if statement: it repeats an operation **while** a condition is true. The syntax of a while-loop looks is as follows:

```

1 while condition:      # HEADER
2     # python code    # BODY

```

⁴Looping is also known as **iteration**

The condition is an expression that evaluates to a boolean value: either True or False. Notice that while is written in lowercase, and **there is a colon after the condition**.

The body of a while-loop is made up of the lines of code that we want to be executed multiple times and, like if-statements, indentation is critical.

We shall introduce this kind of loop through an example. The task is to generate the rows of the table of Centigrade (C) and Fahrenheit (F) values. The C value starts at -20 and is incremented by 5 as long as $C \leq 40$. For each C value we compute the corresponding F value and write out the two temperatures. We postpone to nicely format the C and F columns of numbers and perform for simplicity a plain print C, F statement inside the loop.

```
1 C = -20    # starting value for C
2 dC = 5     # increment of C in loop
3 while C <= 40:    # loop heading with condition
4     F = (9/5)*C + 32    # Conversion from C to F
5     print(C,F)        # Result
6     C = C + dC        # Increment
7 print('Done.')
```

The first statement whose indentation coincides with that of the while line marks the end of the loop and is executed after the loop has terminated. You are encouraged to cut-paste in the code above in a file, run it and observe what happens.

Now, let's consider the following statement:

```
1     C = C + dC    # Increment
```

This is an example of a counter variable or an increment function. It is important to remember that whenever you write a while loop, you must think about an appropriate increment function. Incrementing the value of a variable is frequently done in loops and so there is a special short-hand notation for this and related operations:

```
1 C += dC    # equivalent to C = C + dC
2 C -= dC    # equivalent to C = C - dC
3 C *= dC    # equivalent to C = C*dC
4 C /= dC    # equivalent to C = C/dC
```

If an increment is not defined you may end up with an **infinite loop** - which may be bad. However, programs with infinite loops are not always bad. A robot might be intended to act forever, and the structure of the code might be an infinite loop considering and taking actions. However, unintentional infinite loops are a common programming error, and can have drastic unintended consequences, like causing the user's computer to become unresponsive while all available computation power is used running the loop.

break Statement

Loops **iterate** over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The **break** statement

terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.

```
1 while condition:
2     # some code
3     if condition:
4         break # breaks out of loop
```

Example: Find the first positive integer divisible by both 11 AND 12.

```
1 x = 1
2 while True:
3     if x % 11 == 0 and x % 12 == 0:
4         break
5     x = x + 1
6 print (x, " is divisible by 11 and 12")
```

continue Statement

The **continue** statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

```
1 while condition:
2     # some code
3     if condition:
4         continue # goes back to check while condition
```

Let's say we want to print all natural numbers less than 100 which are *not* multiples of 3 and 5

```
1 x = 1
2 while x <=100:
3     x += 1
4     if x % 3 ==0 or x % 5 == 0:
5         continue
6         #no more code is executed, we go to the next number
7     print(x, end=' ')
```

This is actually not a very elegant solution and has a "mistake". Can you spot the mistake?

3.3.2 for Loops

Definite iteration loops are frequently referred to as **for** loops and exists in nearly all programming languages, including Python. The most basic for loop is a simple numeric range statement with start and end values, something like this:

```
for i = 1 to 10
    <loop body>
```

In Python the for loop is not like the type above. Instead, the for loop iterates over a collection of objects, rather than specifying numeric values or conditions, something like this:

```
for i in <collection>
    <loop body>
```

More formally, the general format of a Python for loop is the following:

```
1 for <var> in <iterable>:
2     <statement(s)>
```

Here <iterable> is a collection of objects⁵. The <statement(s)> in the loop body are denoted by indentation, as with all Python control structures, and are executed once for each item in <iterable>. The loop variable <var> takes on the value of the next element in <iterable> each time through the loop.

A numeric range loop isn't directly built into Python but Python provides a built-in **range** function that can be used to generate a sequence of integers that a for loop can iterate over, as shown below

```
1 x = range(5)
2 print(x)
3 range(0, 5)
4 print(type(x))
5 <class 'range'>
6 # Then one can loop:
7 for i in x:
8     print(x)
9 0
10 1
11 2
12 3
13 4
14 5
```

`range(<begin>, <end>, <stride>)` returns an **iterable** that yields integers starting with <begin>, up to but not including <end>. If specified, <stride> indicates an amount to skip between values (analogous to the stride value used for string and list slicing).

Another iterable object is a string. A string is a collection of characters and can be looped over:

```
1 s = "hello"
2 for c in s:
3     print(c)
4 Out []:
5 p
6 y
7 t
8 h
9 o
10 n
```

Note that the loop prints one character of string `s` in each line.

⁵In Python, iterable means an object can be used in iteration. If an object is iterable, it can be passed to the built-in Python function `iter()`, which returns something called an iterator!

Chapter 4

Lists and Tuples

4.1 Sequence Types

We have introduced Python's built-in data types: `int`, `float`, `str` and `bool`. Now we introduce Python's data structures¹, specifically those known as sequence types. We have already seen one sequence data type: `str`. Sequence types are qualitatively different from numeric types (`int`, `float`) because they are compound data types - meaning they are made up of smaller pieces. Strings, of course, are made up of smaller strings, each containing one character. Another characteristic of sequence data types is that they are **iterable**.

4.2 Lists

A **list** is a **linear data structure**, meaning that its elements have a linear ordering, that can store multiple pieces of information and with a single variable name. The list name, together with a non-negative integer, called the **index**, can then be used to refer to the individual items of data. Finally, a list is a **mutable** data type which means we can change its elements.

4.2.1 Initialization

Lists are enclosed in square brackets (`[` and `]`). These are some examples:

```
1 empty_list = []
2 # or
3 empty_list = list() # list() constructor
4 my_list = [2, 3, 5]
5 shoplister = ['apple', 'mango', 'orange', 'banana']
6 mixed_list = [1, 'a', 3.1416, my_list]
7 a = ['bark', 'meow', 'woof', 'bark', 'cheep', 'bark']
```

Note *mixed_list* contains mixed types, including other lists and can contain much more². The list *a* contains repeated items, showing that list elements need not be unique. Finally, a list can contain

¹As a general rule, data structures are objects that contain a possibly large number of other objects.

²Lists can even contain complex objects, like functions, classes, and modules, which will be discussed later.

any number of objects, from zero to as many as your computer's memory will allow.

4.2.2 Accessing and Editing Lists

Every element in a list is associated with an index, which reflects the position of the element in the list³. Lists in Python use zero-based indexing. Thus, all lists have index values $0 \dots n - 1$, where n is the number of elements in the list.

```
1 my_list = [2, 3, 5]
2 my_list[0] # first element in my_list
3 2
4 my_list[1]
5 3
```

Python allows negative indices, which "count from the right". So, `my_list[-1]` gives the last element of the list `my_list`. `my_list[-2]` is the element before `my_list[-1]`, and so forth.

Elements in lists can be deleted, and new elements can be inserted anywhere. The functionality for doing this is built into the list object and accessed by a **dot notation**⁴.

```
1 my_list.append(7) # adds 7 to end of my_list
2 my_list.insert(1,0) # inserts element 0 in position 1
3 print(my_list)
4 [2, 0, 3, 5, 7]
5 my_list.pop() # removes last element in list
6 print(my_list)
7 [2, 0, 3, 5]
8 del my_list[1] # deletes second element
9 print(my_list)
10 [2, 3, 5]

1 # Since lists are mutable we can do the following
2 my_list[2] = 4
3 print(my_list)
4 [2, 3, 4]
```

4.2.3 Slicing

A subsequence of a sequence is called a slice and the operation that extracts a subsequence is called **slicing**. Like with indexing, we use square brackets (`[]`) as the slice operator, but instead of one integer value inside we have two, separated by a colon (`:`). If a is a list, the expression $a[m : n]$ returns the portion of a from index m to, but not including, index n . For example:

```
1 primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
2 primes[0:3] # yields [2, 3, 5]
3 primes[4:5] # yields [11] - last index is excluded!
4 primes[-3:-1] # yields [23, 29]
```

³Similar to strings, as they are both sequence types.

⁴Recall that in modules we can access module functions using dot notation. A list is defined in a module and so its associated functions can also be accessed using dot notation.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

```
1 primes[0:] # prints the entire sequence
2 primes[9:] # yields [29, 31]
```

You can specify a stride - either positive or negative:

```
1 primes[0:6:2] # Here, 2 is the stride prints
2 [2, 5, 11]
3 primes[6:0:-2]
4 [17, 11, 5]
```

The syntax for reversing a list works the same way it does for strings:

```
1 primes[::-1]
2 [31, 29, 23, 19, 17, 13, 11, 7, 5, 3, 2]
```

4.2.4 Operators

The **in** operator returns whether a given element is contained in a sequence. Example:

```
1 stuff = ['this', 'that', 'these', 'those']
2 'this' in stuff
3 True
```

Notice that **in** works somewhat differently with strings. It evaluates to True if one string is a substring of another. When combined with *not* we get the obvious behavior:

```
1 stuff = ['this', 'that', 'these', 'those']
2 'python' not in stuff
3 True
4 'python' in stuff
5 False
```

The **+** operator is used to denote concatenation. Since the plus sign also denotes addition, Python determines which operation to perform based on the operand types. Thus the plus sign, **+**, is referred to as an **overloaded operator**. If both operands are numeric types, addition is performed. If both operands are sequence types, concatenation is performed. The same applies to *****.

```
1 stuff = ['this', 'that', 'these', 'those']
2 stuff + ['them']
3 ['this', 'that', 'these', 'those', 'them']
4 stuff*2
5 ['this', 'that', 'these', 'those', 'this', 'that', 'these', 'those']
```

Operations **min**/**max** return the smallest/largest value of a sequence, and **sum** returns the sum of all the elements (when of numeric type). **len()** return the length of the sequence. Finally, the comparison operator, **==**, returns True if the two sequences are the same length, and their corresponding elements are equal to each other.

```

1 primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
2 stuff = ['this', 'that', 'these', 'those']
3 min(primes) # 2
4 max(primes) # 31
5 len(primes) # 11
6 len(stuff) # 4
7 min(stuff) # 'that'
8 max(stuff) # 'those'
9 stuff == primes # False

```

4.2.5 Iterating Over List Elements vs. List Index Values

The `for` statement can be applied to all sequence types, including lists.

```

1 # Loop over list elements
2 primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
3 for n in primes:
4     print(n)

```

When the elements of a list need to be accessed, but not altered, a loop variable that iterates over each list element is an appropriate approach. However, there are times when the loop variable must iterate over the *index values* of a list instead.

```

1 nums = [10, 20, 30, 40, 50, 60]
2 # Iterate over elems of list
3 for k in nums:
4     sum = sum + k
5 # Loop over index values
6 for k in range(len(nums)):
7     sum = sum + nums[k]

```

See the difference?

There are situations in which a sequence is to be traversed while a given condition is true. In such cases, a while loop is the appropriate control structure. Let's say that we need to determine whether the value 40 occurs in list *nums* above. In this case, once the value is found, the traversal of the list is terminated.

```

1 nums = [10, 20, 30, 40, 50, 60]
2 k = 0
3 wanted = 40
4 found = False
5 while k < len(nums) and not found:
6     if nums[k] == wanted:
7         found = True
8     else:
9         k += 1

```

4.2.6 List Comprehensions

Because running through a list and for each element creating a new element in another list is a frequently encountered task, Python has a special compact syntax for doing this, called **list comprehension**. The general syntax reads

```
1 newlist = [E(e) for e in list]
```

where $E(e)$ represents an expression involving element e . Here are some examples:

```
1 my_nums = [i*0.5 for i in range(10)]
2 print(my_nums)
3 Out []:
4 [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
```

The "long way" of creating `my_nums` is:

```
1 my_nums = list()
2 for i in range(10):
3     my_nums.append(i*0.5)
```

So, three lines of code versus one. Other examples:

```
1 Cdegrees = [-5 + i*0.5 for i in range(10)]
2 print(Cdegrees)
3 Out []: [-5.0, -4.5, -4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -0.5]
4
5 Fdegrees = [(9.0/5)*C + 32 for C in Cdegrees]
6 print(Fdegrees)
7 Out []: [23.0, 23.9, 24.8, 25.7, 26.6, 27.5, 28.4, 29.3, 30.2, 31.1]
8
9 S = [x**2 for x in range(10)]
10 print(S)
11 Out []: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
12
13 M = [x for x in S if x % 2 == 0]
14 print(M)
15 Out []: [0, 4, 16, 36, 64]
```

Note in the last example we have an **if** condition within the loop. You should become familiar with this notation.

4.2.7 Traversing Multiple Lists Simultaneously

It happens quite frequently that two or more lists need to be traversed simultaneously. As an alternative to the loop over indices, Python offers a special nice syntax that can be sketched as

```
1 for list1, list2, list3... in zip(list1, list2, list3, ...):
2     # work with element e1 from list1, element e2 from list2, etc...
```

The **zip** function turns n lists (`list1, list2, list3, ...`) into one list of n -*tuples*, where each n -*tuple* (`e1,e2,e3,...`) has its first element (`e1`) from the first list (`list1`), the second element (`e2`) from the second list (`list2`), and so forth. The loop stops when the end of the shortest list is reached.

4.2.8 Copying Lists

Because of the way that lists are represented in Python, when a variable is assigned to another variable holding a list, $list2 = list1$, each variable ends up referring to the same *instance* of the list in memory. This is referred to as a **shallow copy** and has important implications. For example, if an element of $list1$ is changed, then the corresponding element of $list2$ will change as well,

```
1 list1 = [10, 20, 30, 40]
2 list2 = list1
3 list1[0] = 5
4 print(list1)
5 [5, 20, 30, 40] # change made in list1
6 print(list2)
7 [5, 20, 30, 40] # change in list1 causes a change in list2
```

Knowing that variables $list1$ and $list2$ refer to the same list explains this behavior. This issue does not apply to strings and tuples, since they are immutable and therefore cannot be modified. When needed, a copy of a list can be made as given below,

```
1 list1 = [10, 20, 30, 40]
2 list2 = list(list1)
3 list1[0] = 5
4 print(list1)
5 [5, 20, 30, 40] # change made in list1
6 print(list2)
7 [10, 20, 30, 40] # change in list1 does NOT cause a change in list2
```

When copying lists that have sublists, another means of copying, called **deep copy**, may be needed.

When a variable is assigned to another variable holding a list, each variable ends up referring to the **same instance** of the list in memory.

Python has a module called *copy* to perform these types of copy operations.

```
1 # importing "copy" for copy operations
2 import copy
3 list1 = [1, 2, [3,5], 4]
4 # using copy to shallow copy
5 shallow_copy = copy.copy(list1)
6 # using deepcopy to deep copy
7 deep_copy = copy.deepcopy(list1)
```

4.3 Tuples

Python provides another sequence type that is an ordered collection of objects, called a tuple. Tuples are identical to lists in **all** respects, except for the following properties:

- Tuples are defined by enclosing the elements in parentheses $()$ instead of square brackets $[]$.

- Tuples are **immutable**.

Examples:

```
1 empty_tuple = ()
2 # or
3 empty_list = tuple() # list() constructor
4 my_tuple = (2, 3, 5, 'cat', 'dog')
5
6 my_tuple[0] = 1
7 Out []:
8 TypeError: 'tuple' object does not support item assignment
```

If we attempt to change my_tuple:

```
1 my_tuple[0] = 1
2 Out []:
3 TypeError: 'tuple' object does not support item assignment
```

showing that tuples are immutable.

Tuples have fewer methods than lists. In fact, the only regular methods are **count** and **index**:

```
1 # List methods:
2 [ 'append',
3   'clear',
4   'copy',
5   'count',
6   'extend',
7   'index',
8   'insert',
9   'pop',
10  'remove',
11  'reverse',
12  'sort']
13 # Tuple methods:
14 ['count',
15  'index']
```

The rest of the list methods are not available for tuple because they modify the object, and tuples, being immutable, cannot be modified.

4.3.1 Why use a tuple instead of a list

- Program execution is faster when manipulating a tuple than it is for the equivalent list.
- Sometimes you don't want data to be modified. If the values in the collection are meant to remain constant for the life of the program, using a tuple instead of a list guards against accidental modification.

- There is another Python data type that you will encounter shortly, called a dictionary, which requires as one of its components a value that is of an immutable type. A tuple can be used for this purpose, whereas a list can't be.

4.3.2 Nested Data Structures

Lists and tuples can contain elements of any type, including other sequences. Thus, lists and tuples can be nested to create arbitrarily complex data structures. Examples:

```
1 class_grades = [ [85, 91, 89], [78, 81, 86], [62, 75, 77]]
2 mixed_nested_list = [ [4, [True, False], 6, 8], [888.0, 999.0] ]
3 tuple_with_list = (1, [2, 3], 4, 5)
```

Note that in the last example the tuple contains the list [2,3]. Lists are mutable and thus can be changed and so we could change [2,3] to, say, [2,4]. Tuples, on the other hand, are immutable. Try it. Does that mean that tuples are not really immutable? How can you show that the tuple is indeed immutable?

Chapter 5

Dictionaries and Sets

5.1 Sets

A mathematical set is a collection of values without duplicates or order. In sets

- Order does not matter, i.e. $\{ 1, 2, 3 \} == \{ 3, 2, 1 \}$
- There are no duplicate entries: $\{ 3, 1, 4, 1, 5 \} == \{ 5, 4, 3, 1 \}$

A Python `set` is an unordered collection of unique and immutable objects.

5.1.1 Properties

- Set elements must be **immutable** values.
- The set itself is **mutable**, that is we can add and remove elements.
- Only set operations change the set
- **Aside:** *frozenset* must contain immutable values and is itself immutable (cannot add and remove elements)

5.1.2 Initialization

There are two ways to initialize sets. The first way is to use direct mathematical syntax with curly braces:

```
1 odd = {1,3,5}
2 # We can also use the default constructor to initialize an empty set
3 empty = set()
```

Unfortunately you cannot use `{}` to express empty set. The second way is to construct a set from a list (or a tuple or a string):

```

1 odd = set([1,3,5])
2 primes = set([2,3,5])
3 fruits = {"apple", "banana", "cherry"}

```

5.1.3 Set operations

Given the sets *odd* and *primes* above we can easily understand set operations:

membership \in	Python: in	4 in primes \Rightarrow False
union \cup	Python: 	odd primes \Rightarrow {1,2,3,5}
intersection \cap	Python: &	odd & primes \Rightarrow {3,5}
difference $-$	Python: -	odd - primes \Rightarrow {1}
symmetric difference	Python: ^	odd ^ primes \Rightarrow {1,2}

Note that we can do iteration over sets

```

1 for n in primes:
2     print(n)

```

But we cannot index into a set to access a specific element

```

1 print[1] # illegal

```

Why? Because sets are unordered.

5.1.4 Modifying a Set

Add element to a set

```

1 primes.add(7)
2 # or
3 primes = primes | {7}

```

Remove element from a set

```

1 primes.remove(2)
2 # or
3 primes = primes - {2}

```

Remove an arbitrary element from a set

```

1 primes.pop()

```

Try the following:

```

1 z = {5, 6, 7, 8}
2 y = {1, 2, 3, "foo", 1, 5}
3 k = z & y
4 j = z | y
5 m = y - z

```



```

6 n = z - y
7 p = z
8 q = set(z)
9 z.add(9)

```

Example: Find the common element in both list1 and list2.

```

1 # Using lists and loops:
2 list1 = [1,2,3,4,5,6,7,8]
3 list2 = [2,4,6,8]
4 out1 = []
5 for i in list2:
6     if i in list1:
7         out1.append(i)
8
9 # Using list comprehensions would reduce the last 4 lines into 1:
10 out1 = [i for i in list2 if i in list1]

```

Using sets you would perform the following operation: set1 & set2:

```

1 list1 = [1,2,3,4,5,6,7,8]
2 list2 = [2,4,6,8]
3 out1 = set(list1) & set(list2)

```

How would you find elements in either list1 or list2 (or both) (without duplicates)? set1 | set2
How would you find elements in either list1 or list2 but not in both? set1 ^ set2

5.2 Dictionaries

So far we have stored information in various types of objects, such as numbers, strings, list, and arrays. A dictionary is a very flexible object for storing various kind of information¹.

A **dictionary** is a mapping that stores keys with associated values.
For every key in a dictionary, there is exactly one value associated with it.

Recall that a list is a collection of objects indexed by an integer going from 0 to the number of elements minus one. Instead of looking up an element through an integer index, it can be more handy to use a text. Roughly speaking, a list where the index can be a text is called a dictionary in Python. Other computer languages use other names for the same thing: HashMap, hash, associative array, or map.

5.2.1 Properties

- Dictionaries are mutable objects.
- Order does not matter.

¹And in particular when reading files, to be discussed later.

- Given a key, can look up a value. Given a value, cannot lookup its key.
- There are no duplicate keys but or or more keys may map to the same value.
- Keys must be immutable

5.2.2 Initialization

Dictionaries are enclosed in curly braces `{}`. There are two ways to initialize them.

```
1 d = {} # an empty dictionary
2 d = dict() # constructor
```

Examples of `dict()` syntax:

```
1 # Temperatures:
2 daily_temps = {'sun': 68.8,
3               'mon': 70.2,
4               'tue': 67.2,
5               'wed': 71.8,
6               'thur': 73.2,
7               'fri': 75.6,
8               'sat': 74.0}
9
10 # Population of cities (in millions)
11 population = {"Chicago":2.7, "New York":8.17, "Rome":2.87,
12              "Paris":2.24, "London":8.78}
13
14 # HTTP response status codes
15 status = {200:"ok", 404:"not found", 400:"bad request"}
16
17 # Keys must be unique but values may be duplicated.
18 id_to_grades = {'A1': 80, 'A2': 90, 'A3': 90}
19
20 # A dictionary can have keys of different types. For example,
21 # one key can be of type int and another of type str:
22 d = {'apple': 1, 3: 4}
```

Note that (key,value) pairs are separated by a colon(`:`).

5.2.3 Dictionary Operations

Let's consider the first example above. Dictionary `daily_temps` stores the average temperature for each day of the week. Each temperature has associated with it a unique key value ('sun', 'mon', etc.). Strings are often used as key values. The syntax for accessing an element of a dictionary is the same as for accessing elements of sequence types, except that a key value is used within the square brackets instead of an index value: `daily_temps['sun']`.

Python dictionaries use index-like notation to refer to the **value** associated with **key** in a dictionary `d`: `d[key]`

Although the elements of the dictionary data structure are physically ordered, the ordering is irrelevant to the way that the structure is utilized. *The location that an element is stored in and retrieved from within such a data structure depends only on its key value*, thus there is no logical first element, second element, and so forth². Let's see how we can use `daily_temps`:

```
1 if daily_temps['sun'] > daily_temps['sat']:
2     print('Sunday was the warmer weekend day')
3 else:
4     if daily_temps['sun'] < daily_temps['sat']:
5         print('Saturday was the warmer weekend day')
6     else:
7         print('Saturday and Sunday were equally warm')
```

Although strings are often used as key values, any immutable type may be used as well, such as a tuple. In this case, the temperature for a specific date is retrieved by,

```
temps[('Feb', 5, 2019)] -> 70.0
```

Note that this key contains both string and integer values.

What follows is a summary some of the basic operations for a dictionary `d`.

<code>d[key]</code>	Get value for key in dictionary.
<code>d[key] = value</code>	Set value for key in dictionary <code>d</code> to be value.
<code>len(d)</code>	Number of key-value pairs in <code>d</code> .
<code>key in d</code>	True if key has an entry in <code>d</code> ; otherwise, False.
<code>key not in d</code>	True if key does not have an entry in <code>d</code> ; otherwise, False.
<code>del d[key]</code>	Delete entry for key in <code>d</code> . Raises <code>KeyError</code> if key is not in <code>d</code> .
<code>sorted(d)</code>	Return sorted list of keys in <code>d</code> . Use <code>sorted(d, key=d.get)</code> to sort the keys by value.

Accessing dictionary elements:

```
1 atomic_number = {"H":1, "Fe":26, "Au":79}
2 atomic_number["Au"] # prints 79
3 atomic_number["B"] # prints KeyError
```

Dictionary functions:

```
1 atomic_number = {"H":1, "Fe":26, "Au":79}
2 atomic_number.keys() # prints dict_keys(['H', 'Fe', 'Au'])
3 atomic_number.values() # prints [1, 79, 26]
4 atomic_number.items() # print dict_items([('H', 1), ('Fe', 26), ('Au', 79)])
```

²The specific location that a value is stored is determined by a particular method of converting key values into index values called **hashing**.

Dictionaries are iterable objects and can therefore be looped over using a for loop:

```
1 atomic_number = {"H":1, "Fe":26, "Au":79}
2
3 # Print out all the keys:
4 for element_name in atomic_number.keys():
5     print(element_name)
6 # Print out all the values:
7 for element_number in atomic_number.values():
8     print(element_number)
9 # Print out the keys and the values
10 for (element_name, element_number) in atomic_number.items():
11     print("name:", element_name, "number:", element_number)
```

Modifying a dictionary:

```
1 us_wars1 = {
2     "Revolutionary" : [1775, 1783],
3     "Mexican" : [1846, 1848],
4     "Civil" : [1861, 1865] }
5 us_wars1["WWI"] = [1917, 1918] # add mapping
6 del us_wars1["Civil"] # remove mapping
```

Inverting a dictionary: How do we reverse key with value in a dictionary? Here's one approach:

```
1 d = {5:25, 6:36, 7:49}
2 k ={}
3 for i in d.keys():
4     k[d[i]] = i
5 print(k)
6 Out []:
7 {25: 5, 36: 6, 49: 7}
```

Try these on your own:

```
1 squares = { 1:1, 2:4, 3:9, 4:16 }
2 squares[3] + squares[3]
3 squares[3 + 3]
4 squares[2] + squares[2]
5 squares[2 + 2]
```

Chapter 6

Functions

So far, we have limited ourselves to using only the most fundamental features of Python - variables, expressions, control structures, and data structures. In theory, these are the only instructions needed to write any program (that is, to perform any computation). From a practical point-of-view, however, these instructions alone are not enough.

The problem is one of complexity. In order to manage the complexity of a large problem, it is broken down into smaller subproblems. Then, each subproblem can be focused on and solved separately. In programming, we do the same thing. Programs are divided into manageable pieces called program routines (or simply routines). Doing so is a form of *abstraction* in which a more general, less detailed view of a system can be achieved. In addition, program routines provide the opportunity for code reuse, so that systems do not have to be created from "scratch". Routines, therefore, are a fundamental building block in software development.

6.1 Functions

In Python program routines are called **functions**. Python functions are similar to the mathematical functions that you are familiar with but they are much more. A function is a collection of statements that you can execute wherever and whenever you want in the program. You may send variables to the function to influence what is getting computed by statements in the function, and the function may return new objects. In particular, functions help to avoid duplicating code snippets by putting all similar snippets in a common place¹. This strategy saves typing and makes it easier to change the program later. Functions are also often used to just split a long program into smaller, more manageable pieces, so the program and your own thinking about it become clearer. Python comes with lots of functions² and we have met several so far.

```
1 # Built-in functions
2 print('hi!')
3 len("hello")
4 pow(2,3)
5 str(17)
```

¹Also known as the Don't Repeat Yourself (DRY principle).

²Built-in functions are always available and do not need to be imported. Other functions come from modules, like **math** or **random**.

```

6 int(input("enter integer: "))
7 range(1,5)
8
9 # functions from modules
10 import math
11 math.sin(math.pi)
12 import random
13 random.random() # this one needs no input

```

6.1.1 Defining Functions

In addition to the built-in or imported functions of Python, there is the capability to define new functions and that makes programming so much more exciting.

A Python **function** is a named group of statements that accomplishes some task.

The elements of a function definition are:

```

1 def a_good_name(arg1, arg2, arg3...): # HEADER
2     # Python statements
3     return value(s) # RETURN VALUE(S)

```

The first line of a function definition is the function header. A function header starts with the keyword **def**, followed by an identifier, which is the function's name³. The function name is followed by a comma-separated (possibly empty) list of **identifiers** (*arg1, arg2, arg3...*) called **parameters**. The actual values passed to the function are called **arguments**. Following the parameter list is a colon (:). Following the function header is the body of the function or program block containing the function's instructions. As with all blocks, the statements must be indented at the same level, relative to the function header.

6.1.2 Docstrings

There is a convention in Python to insert a documentation string right after the **def** line of the function definition. The documentation string, known as a **docstring**, should contain a short description of the purpose of the function and explain what the different arguments and return values are. Interactive sessions from a Python shell are also common to illustrate how the code is used. Docstrings are usually enclosed in triple double quotes (three double-quote " characters), which allow the string to span several lines.

```

1 def line(x0, y0, x1, y1):
2     """
3     Compute the coefficients a and b in the mathematical
4     expression for a straight line y = a*x + b that goes
5     through two points (x0, y0) and (x1, y1).
6     x0, y0: a point on the line (floats).

```

³Function names are important. Be careful of very short names: $f(x)$ is almost always too vague and eventually you will have a hard time knowing or remembering what it does.

```

7     x1, y1: another point on the line (floats).
8     return: coefficients a, b (floats) for the line (y=a*x+b). """
9     a = (y1 - y0)/float(x1 - x0)
10    b = y0 - a*x0
11    return a, b

```

Note that the docstring must appear before any statement in the function body.

The docstring can be accessed in a code as `funcname.__doc__`, where `funcname` is the name of the function, e.g.,

```
1 print(line.__doc__)
```

The `print()` above will print out the documentation of the `line()` function. More importantly, one can use **help** to print the documentation:

```

1 help(line)
2
3 # yields
4
5 Help on function line in module __main__:
6
7 line(x0, y0, x1, y1)
8     Compute the coefficients a and b in the mathematical
9     expression for a straight line y = a*x + b that goes
10    through two points (x0, y0) and (x1, y1).
11    x0, y0: a point on the line (floats).
12    x1, y1: another point on the line (floats).
13    return: coefficients a, b (floats) for the line (y=a*x+b).

```

6.1.3 Value-Returning Functions

A value-returning function is one called for its return value, and is therefore similar to a mathematical function. Take the simple mathematical function $f(x) = 2x$. In this notation, x stands for any numeric value that function f may be applied to, for example, $f(2) = 2(2) = 4$:

```

1 def f(x): # x is a parameter
2     return 2*x

```

Note there is a keyword **return** to specify what result to return. After **return** we have an expression that holds the returned value.

Of course Python functions may return more than one value. So, if we are interested in writing a function that returns the roots of a quadratic equation then we will need to return two values.

6.1.4 Non-Value-Returning Functions

A non-value-returning function is called not for a returned value, but for its **side effects**. A side effect is an action other than returning a function value, such as displaying output on the screen.

```

1 def print_hello():
2     print("Hello, world")

```

Note that `print_hello()` has an empty parameter list and no return statement.

6.1.5 Calling Functions

Every function has the capacity to perform a task, but it only performs that task when it is called. A function call requests execution of the function with particular arguments passed as the values for its parameters (if any). The syntax of a function call is:

```

1 output = name(arg1, arg2, ...)

```

When this expression appears in a program statement that is being executed, the function called *name* executes, using the argument values inside parentheses as the values of its parameters and returns the values specified inside the function: *output* above must match the number of values returned by the functions. Examples:

```

1 def square(x): # x is a parameter
2     return x * x
3 sq = square(2) # 2 is an argument, function returns sq=4
4
5 def my_sum(x, y):
6     return x + y
7 sum = my_sum(2,3) # returns sum=5
8
9 def quadratic_roots(a, b, c):
10    # some code to solve quadratic formula
11    return x1, x2
12 root1, root2 = quadratic_roots(1, -1, 1)

```

Note that in the last function, `quadratic_roots`, we need two values on the left-hand side of the assignment operator because the function returns two values

Note that there is a fundamental difference in the way that value-returning and non-value-returning functions are called:

```

1 def print_hello():
2     print("Hello, world")
3
4 # calling print_hello()
5 print_hello()
6 Out []:
7 Hello, world

```

Although `print_hello()` does not return a value and thus there is no left-hand-side (no variable assigned to the returned value), there is still a value returned! That value is **None**. **None** defines a **null** object and is the default return value in functions without a return statement. To see this effect we can attempt to print the function call itself:

```

1 print(print_hello())

```



```
2 Out []:  
3 Hello, world  
4 None
```

6.1.6 Parameter Passing

Parameter passing is the process of passing arguments to a function. As we have seen, actual arguments are the values passed to a function's formal parameters to be operated on. For example, consider the following program (you should run it):

```
1 def ordered(n1, n2): # n1, n2 are parameters  
2     return n1 < n2 # returns either True or False  
3  
4 num1 = int(input('Enter num1: '))  
5 num2 = int(input('Enter num2: '))  
6  
7 if ordered(num1, num2): # num1, num2 are arguments  
8     print('First number is smaller')  
9 else:  
10    if ordered(num2, num1):  
11        print('First number is larger')  
12    else:  
13        print('Numbers are equal')
```

In this example, function `ordered` is called once with arguments `num1`, `num2` and a second time with arguments `num2`, `num1`. There is one important observation:

The correspondence of arguments and parameters is determined by the **order** of the arguments passed, and not their names.

6.1.7 Keyword Arguments

The functions we have looked at so far were called with a fixed number of **positional arguments**. A positional argument is an argument that is assigned to a particular parameter based on its position in the argument list, as shown below.

```
1 def mortgage_rate(amount, rate, term):  
2     # some calculations  
3     return some_value  
4 monthly_payment = mortgage_rate(350000, 0.06, 20)
```

This function computes and returns the monthly mortgage payment for a given loan amount (`amount`), interest rate (`rate`), and number of years of the loan (`term`).

Python provides the option of calling any function by the use of **keyword arguments**. A keyword argument is an argument that is specified by parameter name, rather than as a positional argument as shown below:

```
1 monthly_payment = mortgage_rate(rate=0.06, term=20, amount=350000)
```

This can be a useful way of calling a function if it is easier to remember the parameter names than it is to remember their order. It is possible to call a function with the use of both positional and keyword arguments. However, all positional arguments must come before all keyword arguments in the function call, as shown below:

```
1 monthly_payment = mortgage_rate(350000, rate=0.06, term=20)
```

This form of function call might be useful, for example, if you remember that the first argument is the loan amount, but you are not sure of the order of the last two arguments rate and term.

6.1.8 Default Arguments

Python also provides the ability to assign a default value to any function parameter allowing for the use of **default arguments**. A default argument is an argument that can be optionally provided, as shown here:

```
1 def mortgage_rate(amount, rate, term=20):
2     # some calculations
3     return some_value
4 monthly_payment = mortgage_rate(350000, 0.06)
```

In this case, the third argument in calls to function mortgage_rate is optional. If omitted, parameter term will default to the value 20 (years) as shown. If, on the other hand, a third argument is provided, the value passed replaces the default parameter value.

Now try the following making sure you understand how the program works:

```
1 def addup(first, last, incr=-1):
2     if first > last:
3         sum = -1
4     else:
5         sum = 0
6         for i in range(first, last+1, incr):
7             sum = sum + i
8     return sum
9 addup(1, 10)
10 addup(1, 10, 2)
11 addup(first=-1, last = -10)
12 addup(incr=-2, first=-1, last = -10)
```

6.1.9 Variable Scope

A **local variable** is a variable that is only accessible from within a given function. Such variables are said to have **local scope**. In Python, any variable assigned a value in a function becomes a local variable of the function. Consider the following

```
1 def func1():
2     n = 10
3     print('func1 ',n)
```

```

4 def func2():
5     n = 20
6     print('func2 ',n)
7     func1()
8     print('func2 ',n)
9 func2()
10 Out []:
11 func2  20
12 func1  10
13 func2  20

```

Both `func1` and `func2` contain identifier `n`. Function `func1` assigns `n` to 10, while function `func2` assigns `n` to 20. Both functions display the value of `n` when called - `func2` displays the value of `n` both before and after its call to `func1`. If identifier `n` represents the same variable, then shouldn't its value change to 10 after the call to `func1`? However, as shown by the output, the value of `n` remains 20. This is because there are two distinct instances of variable `n`, each local to the function assigned in and inaccessible from the other.

Now try commenting out `n = 10` in `func1()` and re-run the above. In that case you will get an error indicating that variable `n` is not defined within `func1`. This is because variable `n` defined in `func2` is inaccessible from `func1`. (In this case, `n` is expected to be a **global variable**).

The period of time that a variable exists is called its **lifetime**. Local variables are automatically created (allocated memory) when a function is called, and destroyed (deallocated) when the function terminates. Thus, the lifetime of a local variable is equal to the duration of its function's execution. Consequently, the values of local variables are not retained from one function call to the next.

The concept of a local variable is an important one in programming. It allows variables to be defined in a function without regard to the variable names used in other functions of the program. It also allows previously written functions to be easily incorporated into a program. The use of global variables, on the other hand, brings potential havoc to programs⁴.

6.1.10 Functions as Arguments to Functions

One frequently needs to have functions as arguments in other functions. For example:

```

1 from math import log, e
2 def silly_func(f, x):
3     return f(x)
4 def func(x):
5     return silly_func(math.log, x)
6 func(math.e)
7 Out []:
8 1.0

```

This is indeed a silly example just to illustrate how a function, in this case the natural log, can be passed to another function.

⁴For this reason, the use of global variables is generally considered to be bad programming style.

A Calculus Example

More commonly, we need Python functions as arguments whenever we do the following:

1. numerical root finding: solve $f(x) = 0$ approximately
2. numerical differentiation: compute $f'(x) = 0$ approximately
3. numerical integration: compute $\int_a^b f(x)dx$ approximately
4. numerical solution of differential equations: $\frac{dx}{dy} = f(x)$

In such Python functions we need to have the $f(x)$ function as an argument f . For example, consider a function for computing the second-order derivative of a function $f(x)$ numerically:

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2} \quad (6.1)$$

where h is a small number. The approximation 6.1 becomes exact in the limit $h \rightarrow 0$. A Python function for $f''(x)$ can be implemented as follows:

```
1 def diff2(f, x, h=1E-6):
2     """
3     approximation of the second-order derivative of a function
4     """
5     r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
6     return r
```

The f argument is like any other argument, i.e., a name for an object, here a function object that we can call as we normally call function objects. An application of `diff2` can read

```
1 # This is the function that we want to take the derivative of
2 def g(t):
3     return t**3
4 t = 1.0
5 d2g = diff2(g, t)
6 print("g({})={:.8f}".format(t, d2g))
7 # prints
8 g(1.0)=5.9996 # Note the exact value is 6.0
```

Now, we know that the approximation formula 6.1 becomes more accurate as h decreases. Let's try to show this:

```
1 for k in range(1,15):
2     h = 10**(-k) # we will decrease the value of h
3     d2g = diff2(g, 1, h)
4     print("h = {}, g({})={:.8f}".format(h, t, d2g))
5 # prints:
6 h = 0.1, g(1.0)=6.00000000
7 h = 0.01, g(1.0)=6.00000000
8 h = 0.001, g(1.0)=6.00000000
```

```

9 h = 0.0001, g(1.0)=5.99999999
10 h = 1e-05, g(1.0)=6.00000272
11 h = 1e-06, g(1.0)=5.99964523
12 h = 1e-07, g(1.0)=6.03961325
13 h = 1e-08, g(1.0)=2.22044605
14 h = 1e-09, g(1.0)=444.08920985
15 h = 1e-10, g(1.0)=0.00000000
16 h = 1e-11, g(1.0)=0.00000000
17 h = 1e-12, g(1.0)=444089209.85006267
18 h = 1e-13, g(1.0)=-44408920985.00625610
19 h = 1e-14, g(1.0)=0.00000000

```

The mathematical result in equation 6.1 gives a better approximation to $f''(x)$ as h gets smaller and smaller but for $h < 10^{-8}$ our computational results are totally wrong. The problem is that for small h , round-off errors in the formula 6.1 blow up and destroy the accuracy. Or more precisely, the result holds until h in the present case reaches 10^{-6} .

The reason for the inaccuracy is that the numerator in 6.1 contains subtraction of quantities that are almost equal. The result is a very small and inaccurate number. The inaccuracy is magnified by h^{-2} , a number that becomes large for small h . Switching from the standard floating-point numbers (float) to numbers with arbitrary high precision resolves the problem. Python has a module **decimal** that can be used for this purpose.

6.1.11 The Main Program

In programs containing functions we often refer to a part of the program as the **main program**. This is the collection of all the statements outside the functions, plus the definition of all functions. Let us look at a complete program:

```

1 from math import exp, sin, pi # in main
2 def f(x):                      # in main
3     e = exp(-0.1*x)
4     s = sin(6*pi*x)
5     return e*s
6 x= 2                            # in main
7 y= f(x)                         # in main
8 print('f({:.4f})={:.4f}'.format(x, y)) # in main

```

The main program here consists of the lines with the comment "in main". The execution always starts with the first line in the main program. When a function is encountered, its statements are just used to define the function - nothing gets computed inside the function before we explicitly call the function, either from the main program or from another function. All variables initialized in the main program become **global variables**. One should always avoid global variables. In our example we can do that by defining yet another function called **main** as follows:

```

1 from math import exp, sin, pi
2 def f(x):
3     e = exp(-0.1*x)
4     s = sin(6*pi*x)
5     return e*s

```

```

6 def main():
7     x= 2
8     y= f(x)
9     print('f({:.4f})={:.4f}'.format(x, y))
10 main()
11
12 Out []:
13 f(2.0000)=-0.0000

```

Note that the main program is also a function which we can call like any other function. This approach can be very useful when creating larger scripts.

6.1.12 Lambda Functions

There is a quick one-line construction of functions that is sometimes convenient:

```
1 f = lambda x: x**2 + 4
```

This **lambda** function is equivalent to writing

```
1 def f:
2     return x**2 + 4

```

In general,

```
1 def g(arg1, arg2, arg3, ...):
2     return expression

```

can be written as

```
1 g = lambda arg1, arg2, arg3, ...: expression
```

Lambda functions are generally used to quickly define a function as argument to another function. Because lambda functions can be defined "on the fly" and thereby save typing of a separate function with *def* and an intended block, lambda functions are popular among many programmers.

6.2 A Bioinformatics Example

The genetic code of all living organisms are represented by a long sequence of simple molecules called nucleotides, or bases, which makes up the Deoxyribonucleic acid, better known as DNA. There are only four such nucleotides, and the entire genetic code of a human can be seen as a simple, though 3 billion long, string of the letters A, C, G, and T. Analyzing DNA data to gain increased biological understanding is much about searching in (long) strings for certain string patterns involving the letters A, C, G, and T. This is an integral part of bioinformatics, a scientific discipline addressing the use of computers to search for, explore, and use information about genes, nucleic acids, and proteins.

6.2.1 Counting Letters in DNA Strings

Given some string **dna** containing the letters A, C, G, or T, representing the bases that make up DNA, we ask the question: how many times does a certain base occur in the DNA string? For example, if **dna** is **ATGGCATT** then we ask how many times the base **A** occurs in this string. In the simple case the answer is 3.

How can we implement this in Python? The most straightforward solution is to loop over the letters in the string, test if the current letter equals the desired one, and if so, increase a counter. Looping over the letters is obvious if the letters are stored in a list. This is easily done by converting a string to a list:

```
1 def count_str0(dna, base):
2     dna = list(dna) # convert string to list of letters
3     i = 0 # counter
4     for c in dna:
5         if c == base:
6             i += 1
7     return i
8 dna = "ATGCGGACCTAT"
9 base = "C"
10 count = count_str0(dna, base)
11 print("{} appears {:d} times in {}".format(base, n, dna))
```

NOTE It is fundamental for correct programming to understand how to simulate a program by hand, statement by statement. Two tools are effective for helping you reach the required understanding of performing a simulation by hand: (i) printing variables, (ii) using a debugger.⁵ You may have noticed that converting the string **dna** to a list is actually unnecessary as we can just iterate over the string (after all it is a sequence):

```
1 def count_str1(dna, base):
2     i = 0 # counter
3     for c in dna:
4         if c == base:
5             i += 1
6     return i
7 dna = "ATGCGGACCTAT"
8 base = "C"
9 count = count_str1(dna, base)
10 print("{} appears {:d} times in {}".format(base, n, dna))
```

The same problem can be solved using a for loop to iterate over the DNA string:

```
1 def count_str2(dna, base):
2     i = 0 # counter
3     for j in range(len(dna)):
4         if dna[j] == base:
5             i += 1
6     return i
```

⁵The Python Tutor can be, at least for small programs, a splendid alternative to debuggers.

```

7 dna = "ATGCGGACCTAT"
8 base = "C"
9 count = count_str2(dna , base)
10 print("{} appears {:d} times in {}".format(base, count, dna))

```

Do you see the difference with the earlier solution? Can you think of another way to solve the problem (perhaps using a while loop)?

Note that a common theme in the `count_str` algorithms is that we need to check when the letter we search for is found in the DNA string. Thus, the idea could be to create a list *found* where each element is True if the base is found in the DNA string. The number of True values, i.e. the length of the list, is the number of letters of the base in the DNA. Consider the following:

```

1 def count_str3(dna, base):
2     found = [] # matches for base in dna: found[i]=True if dna[i]==
                 base for c in dna:
3     for c in dna:
4         if c == base:
5             found.append(True)
6         else:
7             found.append(False)
8     return sum(found) # note we are using sum here
9 dna = "ATGCGGACCTAT"
10 base = "C"
11 count = count_str3(dna , base)
12 print("{} appears {:d} times in {}".format(base, count, dna))

```

Finally we can use boolean values directly, as follows:

```

1 def count_str4(dna, base):
2     found = []
3     for c in dna:
4         found.append(c == base)
5     return sum(found) # note we are using sum here
6 dna = "ATGCGGACCTAT"
7 base = "C"
8 count = count_str4(dna , base)
9 print("{} appears {:d} times in {}".format(base, count, dna))

```

Finally, let's try Python's library.

```

1 def count_str5(dna, base):
2     return dna.count(base)
3 dna = "ATGCGGACCTAT"
4 base = "C"
5 count = count_str5(dna , base)
6 print("{} appears {:d} times in {}".format(base, count, dna))

```

There are probably a few other ways to solve this problem. There are two lessons here:

- There may be multiple ways, i.e. algorithms, to solve a problem.

- Depending on the problem, you may want to choose the most efficient algorithm.

Deciding what constitutes an efficient algorithm is beyond the scope of this course. However, we can explore one measure of efficiency: the CPU time, i.e. which one of the above implementations is the fastest? To answer the question we need some test data, which should be a huge string of DNA.

Generating Random DNA Strings. The simplest way of generating a long string is to repeat a character a large number of times:

```
1 N=1000000
2 dna = 'A'*N
```

The resulting string is just AAA...A, of length N, which is fine for testing the efficiency of Python functions. Nevertheless, it is more exciting to work with a DNA string with letters from the whole alphabet A, C, G, and T. To make a DNA string with a random composition of the letters we can first make a list of random letters and then join all those letters to a string:

```
1 import random
2
3 def generate_string(N, alphabet='ACGT'):
4     my_list = []
5     for i in range(N):
6         my_list.append(random.choice(alphabet))
7     return ''.join(my_list) # this returns a string
8 # test
9 print(generate_string(10)) # will be "random"
10 dna = generate_string(10000000) # 1e7 letters
```

In the snippet above, the `random.choice(alphabet)` function selects an element in the list `alphabet` at random. The `join` function is used to join the elements of the list into a string (see `help(dna.join)` for more information).

Measuring CPU Time. Our next goal is to see how much time the various `count_*` functions spend on counting letters in a huge string, which is to be generated as shown above. Measuring the time spent in a program can be done by the `time` module:

```
1 import time
2 ...
3 t0 = time.clock()
4 # do stuff
5 t1 = time.clock()
6 cpu_time = t1 - t0
```

The `time.clock()` function returns the CPU time spent in the program since its start. If the interest is in the total time, also including reading and writing files, `time.time()` is the appropriate function to call. For small bits of code one can use the `timeit` module. Running through all our functions made so far and recording timings can be done as follows:

```
1 import time
2 functions = [count_str0,
3             count_str1,
```

```

4     count_str2,
5     count_str3,
6     count_str4,
7     count_str5]
8 timings = [] # timings[i] holds CPU time for functions[i]
9 for function in functions:
10    t0 = time.process_time()
11    function(dna, 'A')
12    t1 = time.process_time()
13    cpu_time = t1 - t0
14    timings.append(cpu_time)

```

In Python, functions are ordinary objects so making a list of functions is no more special than making a list of strings or numbers.

We can now iterate over timings and functions simultaneously via zip to make a nice printout of the results:

```

1 for cpu_time, function in zip(timings, functions):
2     print ('{f:<9s}: {cpu:.2f} s'.format(f=function.__name__, cpu=
        cpu_time))
3
4 # Timings:
5 count_str0: 0.80 s
6 count_str1: 0.72 s
7 count_str2: 1.27 s
8 count_str3: 1.89 s
9 count_str4: 1.80 s
10 count_str5: 0.05 s

```

It looks like count_str1 - the simple iteration over the string - is the best performer of all the user-defined functions. However, the built-in count functionality of strings (dna.count(base) in count_str5) runs 14 times faster than the best of the user defined Python functions! The reason is that the for loop needed to count in dna.count(base) is actually implemented in C and runs very much faster than loops in Python.

The message is: use the built-in functions if possible.

Chapter 7

File IO

Input data for a program often comes from files and the results of the computations are often written to files. We will discuss how Python can access information in files and how to create new files. We will focus mainly on text files, that is files that contain data readable by humans. Files that hold non-text data, also called binary files, are commonly used over text files to pack data much more densely and provide much faster access. We will only consider binary files briefly in this course.

7.1 Reading from a text file

Consider the following directory structure:

```
| ____cds-230
| | ____scripts
| | ____data
| | ____homework
| | ____docs
| | ____lectures
```

Suppose we have some measurements in a data file, call it **data1.txt**, stored in the **data** folder. Our goal is to read the the measurement values in **data1.txt**, find the average value, and print out the result.

Before trying to let a program read a file, we must know the file format, i.e., what the contents of the file look like, because the structure of the text in the file greatly influences the set of statements needed to read the file. We therefore start with viewing the contents of the file `data1.txt`. To this end, load the file into a text editor or viewer. What we see is a column with numbers:

```
21.8
18.1
19
23
26
17.8
```

Our task is to read this column of numbers into a list in the program and compute the average of the list items. To read from a text file, follow this basic paradigm:

- Open the file for reading.
- Read the line from the file.
- Process the line. In other words, do whatever you need to do with the line.
- Close the file.

To open a file for reading, call the built-in Python function **open**. It takes two parameters, both strings. The first parameter gives the name of the file. The second parameter should be the string "r", which indicates that you are reading from the file (if absent then "r" is assumed). The function returns a reference to an *object* representing the file.

```
1 file_object = open(file_name, "r")
```

For example, to open the file data1.txt for reading - assuming the file is in the same directory where we are running the Python code:

```
1 file_name = "data1.txt"
2 in_file = open(file_name, "r")
```

When the location of a file is in another location, say in the **data** directory, then you will need to specify more than the file name, you will need to specify the **path** of the file. Depending on your operating system the paths may look different. For example:

1. Linux/Mac: `"/Users/jdoe/cds-230/data/data1.txt"`
2. Linux/Mac: `"data/data1.txt"`
3. Windows: `"C:\Users\jdoe\My Documents\cds-230\data\data1.txt"`
4. Windows: `"\data\data1.txt"`

An **absolute** filename gives a specific location on disk as in (1) or (3) above. A **relative** filename gives a location relative to the current working directory as in (2) or (4) above¹. **Warning:** code will fail with a **FileNotFoundError** error if it fails to find the file specified in file_name.

So, assuming we are in the cds-230 directory:

```
1 file_name = "data/data1.txt"
2 in_file = open(file_name, "r")
```

Of course, you could get that down to a single line:

```
1 in_file = open("data/data1.txt", "r")
2 print(type(in_file))
3 <class '_io.TextIOWrapper'>
```

¹A relative filename is usually a better choice

To read lines from the file, there are two general approaches:

Process entire file at once:

```
1 all_data_in_one_big_string = in_file.read()
2 print(all_data_in_one_big_string)
3 21.8
4 18.1
5 19
6 23
7 26
8 17.8
```

This approach can be very useful when you are performing operations on the entire text. We will get back to this later.

Process one line at a time using a for-loop²

```
1 for line in in_file:
2     # Do something with line, e.g. print(line)
```

Here, in each iteration of the loop, the variable `line` will hold the current line from the file. *Notice that this type of for-loop differs from the for-loops we have already seen, in that what follows in is not a list, but a reference to a file object.*

In a file, each line ends with a newline character, which we indicate in Python by `"\n"`. Each line that you read from the file will be a string containing all the characters in the line, including the newline character. Note that the newline character, `"\n"`, is a **non-printing character**.

After the file is read, one should close the file object with `close()`.

```
1 in_file.close()
```

Instead of reading one line at a time, we can load all lines into a list of strings (`lines`) by (make sure you reopen the file):

```
1 lines = in_file.readlines()
2 print(lines)
3 ['21.8\n', '18.1\n', '19\n', '23\n', '26\n', '17.8\n']
```

Note that the output from `readlines()` allows you to "see" the `"\n"` characters. Using `readlines()` is equivalent to:

```
1 lines = []
2 for line in in_file:
3     lines.append(line)
```

or the list comprehension:

```
1 lines = [line for line in in_file]
```

²Assumption: file is a **sequence** of lines, i.e. strings.

It is important to emphasize that **once you have read a file, the file pointer will be positioned at the end of the file** (also known as EOF). Therefore any subsequent read operation will yield nothing. In that case you will want to *rewind* the file. In that case you can use the `seek()` file function as follows:

```
1 # open a file
2 # read a file ...
3 # ... I need to read the file again, so rewind
4 in_file.seek(0)
```

Here `seek(0)` is moving the file pointer back to the first position, i.e. 0, thus allowing you to read the file from the start. You can also use `seek()` to move to any position in the file.

Let's get back to our task. Once we have the data loaded into a we can compute the average of the numbers in the file. One way to do this is as follows:

```
1 mean = 0
2 for number in lines: # lines contain numbers
3     mean = mean + float(number) # convert number to float
4 mean = mean/len(lines)
5 print(mean)
6 20.95
```

There is one important thing to note in the for loop: `float(number)`. Python's reading functions return **text**, i.e. *str* objects. But since we need to perform numeric calculations we must convert the *str* objects, each line in the list, into numbers by using type casting.

Summing up a list of numbers is often done in numerical programs, so Python has a special function called **sum** for performing this task. However, `sum` must in the present case operate on a list of floats, not strings. We can use a list comprehension to turn all elements in `lines` into corresponding float objects:

```
1 mean = sum([float(line) for line in lines])/len(lines)
```

An alternative implementation is to load the lines into a list of float objects directly.

```
1 in_file = open('data/data1.txt')
2 numbers = [float(line) for line in in_file.readlines()]
3 in_file.close()
4 mean = sum(numbers)/len(numbers)
5 print(mean)
6 20.95
```

It is probably clear -or confusing - by now that one problem is solved by many alternative sets of statements, but this is the very nature of programming. Once we have gained experience we can judge several alternative solutions to a programming task and choose one that is either particularly compact, easy to understand, and/or easy to extend later.

7.2 Reading a Mixture of Text and Numbers

The `data1.txt` file has a very simple structure since it contains numbers only. Many data files contain a mix of text and numbers. The file `data2.txt` provides an example (see <http://www>.

worldclimate.com/cgi-bin/data.pl?ref=N41E012+2100+1623501G1):

```
Average rainfall (in mm) in Rome: 1188 months between 1782 and 1970
Jan 81.2
Feb 63.2
Mar 70.3
Apr 55.7
May 53.0
Jun 36.4
Jul 17.5
Aug 27.5
Sep 60.9
Oct 117.7
Nov 111.0
Dec 97.9
Year 792.9
```

How can we read the rainfall data in this file and extract the values?

The most straightforward solution is to read the file line by line, and for each line split the line into words, pick out the last (second) word on the line, convert this word to float, and store the float objects in a list. The complete code, wrapped in a function, may look like this

```
1 def extract_data(filename):
2     in_file = open(filename)
3     in_file.readline() # skip the first line
4     numbers = []
5     for line in in_file:
6         words = line.split()
7         number = float(words[1])
8         numbers.append(number)
9     in_file.close()
10    return numbers
11 values = extract_data('data2.txt')
12 print(values)
13 [63.2, 70.3, 55.7, 53.0, 36.4, 17.5, 27.5, 60.9, 117.7, 111.0, 97.9,
    792.9]
```

Note that the first line in the file is just a comment line and of no interest to us. We therefore read this line by `in_file.readline()`. The for loop over the lines in the file will then start from the next (second) line.

Can you think of a way to condense the program above?

7.3 With statements

Because things can go wrong when opening a file - for example, it may not exist or it may be corrupted - it is important to open files carefully so that errors may be handled gracefully. The *recommended* way to open a file in Python is to use a **with** statement. Its syntax is the following:

```
1 with expression as variable:
2     # body
```

What happens when this statement executes is complicated, but when it is used to open a file, the steps are approximately like this:

1. An attempt is made to open the file.
2. If successful, the open file object is assigned to variable.
3. The body is executed.
4. The file is closed at the end, even if something goes wrong in the body.

The above example would be written as follows:

```
1 file_name = "data1.txt"
2 with open(file_name) as in_file:
3     for line in in_file:
4         print(line)
```

You can do all the usual file I/O operations that you would normally do as long as you are within the **with** code block. Once you leave that code block, the file handle will close and you will not be able to use it any more. You no longer have to close the file handle explicitly as the **with** operator does it automatically! See if you can change some of the earlier examples from this chapter so that they use the with method too.

7.4 Writing to a text file

Writing to a file is similar to reading from a file, but it differs in some important ways.

First, you have to open the file for writing. The second parameter to the open function should not be "r". Instead, you have your choice of "w" or "a". If you want to append to a file - that is, add to the end of an existing file - then choose "a". If you want to overwrite the file if it already exists, choose "w". If the file does not exist and you choose "a", you get the same effect as choosing "w". So, for example:

```
1 out_file = open(file_name, "w")
```

There is nothing special about the name out_file; use whatever name you like.

Just as you close a file that you are reading when you are done reading from it, you must close a file that you are writing when you are done writing to it.

To write to a file, call the **write** method on the file. This method takes one parameter, a string, and it writes to the end of the file. If you want a newline in the file, you have to put it into the string yourself.

Here is an example of writing a file:


```

1 out_file = open("gettysburg.txt", "w")
2 out_file.write("Four score and seven years ago,\n")
3 out_file.write("our fathers brought forth on this continent, a new
  nation,\n")
4 out_file.write("conceived in Liberty, and dedicated to the
  proposition ")
5 out_file.write("that all men are created equal.\n")
6 out_file.close()

```

As another example consider writing a table to a file. Suppose we have an array of data

```

1 data = [[ 0.75, 0.29619813, -0.29619813, -0.75 ],
2         [ 0.29619813, 0.11697778, -0.11697778, -0.29619813],
3         [-0.29619813, -0.11697778, 0.11697778, 0.29619813],
4         [-0.75, -0.29619813, 0.29619813, 0.75 ]]

```

We iterate through the rows (first index) in the list, and for each row, we iterate through the column values (second index) and write each value to the file. At the end of each row, we must insert a newline character in the file to get a linebreak.

```

1 out_file = open('my_table.dat', 'w')
2 for row in data:
3     for column in row:
4         out_file.write('{:14.8f}'.format(column))
5     out_file.write('\n')
6 out_file.close()

```

The resulting data file becomes:

```

0.75000000    0.29619813   -0.29619813   -0.75000000
0.29619813    0.11697778   -0.11697778   -0.29619813
-0.29619813   -0.11697778    0.11697778    0.29619813
-0.75000000   -0.29619813    0.29619813    0.75000000

```

7.5 Binary Files

Files that hold photographs, videos, zip files, executable programs, etc. are called binary files: they are not organized into lines, and cannot be opened with a normal text editor. However, Python works just as easily with binary files, but when we read from the file we are going to get bytes back rather than a string. Furthermore, you will find that the contents of a binary file will be, upon printing, unreadable.

In Python, the binary data is represented using a special type called **bytes**. The **bytes** type represents an immutable sequence of numbers between 0 and 255.

To try to understand this, let's create a binary version of a simple text file and then try to read it. In Python reading and writing a binary file is done by appending **b** to the mode parameter:

```

1 with open('hello.bin', 'wb') as f:
2     str = b'hello world!' # note the 'b' prefix

```

```

3     f.write(str)
4
5 with open('hello.bin', 'rb') as f:
6     print(f.read())
7 b'hello world!' # note the 'b' prefix

```

Note that the input to `write()` must be of **bytes** type, i.e. to create a bytes-like object from a string we prefix the string with a *b*. Similarly, the output from `read` is in **bytes** and so Python prefixes a *b* to the string. The *b* means that those are byte objects. Since bytes are just 1s and 0s we need to to **decode** or **encode** these objects whenever we read or write in binary mode.

Encoding³ and decoding⁴ refer to an intermediate representation of the text process so that we can work with data on a computer. After all data are internally represented as 1s and 0s but what we get when we read a file is numbers and text. There are several encodings and Python uses one called **UTF-8**. For the `hello.bin` data file we do the following:

```

1 with open('hello.bin', 'rb') as f:
2     data = f.read()
3     text = data.decode('utf-8') # from binary to text
4     print(text)
5 Out []:
6 hello world! # note that there is no 'b'
7
8 with open('hello.bin', 'wb') as f:
9     text = 'hello world!'
10    f.write(text.encode('utf-8')) #from text to binary

```

The `encode` and `decode` functions accept a few parameters. The first and most important one, is the parameter that will indicate the encoding system. As shown above we used `'utf-8'` to decode data from binary to text and also `'utf-8'` to encode from text to binary.

Finally, it is important to note that, in our case, binary data happens to contain printable characters, like alphabets, newline, etc. However, this will not be the case most of the time. It means that with binary data we cannot reliably use `readline()` and file object (as an iterator) to read the contents of a file because might be no newline character in a file. The best way to read binary data is to **read it in chunks** using the `read()` method.

7.6 More on encoding and decoding: Unicode characters

If you find yourself dealing with text that contains non-ASCII⁵ characters, you have to learn about **Unicode** - what it is, how it works, and how Python uses it.

Learning about Unicode is beyond the scope of this class. However, we can try to get a general idea as to how Python deals with it. First, you must understand the difference between bytes and characters. In older, ASCII-centric languages and environments, bytes and characters are treated as the same thing. Since a byte can hold up to 256 values, these environments are limited to 256

³The process of turning abstract symbols into bits

⁴The process of reading bits (1s and 0s), making sense out of them, and getting back symbols or characters

⁵Remember that the original ASCII character set encodes 128 specified characters into seven-bit integers and can be used to represent all written text used in the English language.

characters. Unicode, on the other hand, has tens of thousands of characters. That means that each Unicode character takes more than one byte, so you need to make the distinction between characters and bytes.

Standard Python strings are really byte strings, and a Python character is really a byte. Other terms for the standard Python type are "8-bit string" and "plain string."

Conversely, a Python Unicode character is an abstract object big enough to hold the character, analogous to Python's long integers. You don't have to worry about the internal representation; the representation of Unicode characters becomes an issue only when you are trying to send them to some byte-oriented function, such as the `write` method for files. At that point, you must choose how to represent the characters as bytes. Converting from Unicode to a byte string is called **encoding** the string. Similarly, when you load Unicode strings from a file or other byte-oriented object, you need to **decode** the strings from bytes to characters.

There are many ways of converting Unicode objects to byte strings, each of which is called an encoding. For a variety of historical, political, and technical reasons, there is no one "right" encoding. Every encoding has a case-insensitive name, and that name is passed to the `decode` method as a parameter. Here are a few you should know about:

- The **UTF-8** encoding can handle any Unicode character. It is also backward compatible with ASCII, so a pure ASCII file can also be considered a UTF-8 file, and a UTF-8 file that happens to use only ASCII characters is identical to an ASCII file with the same characters. This property makes UTF-8 very backward-compatible, especially with older Unix tools. UTF-8 is far and away the dominant encoding on Unix. Its primary weakness is that it is fairly inefficient for Eastern texts.
- The **UTF-16** encoding is favored by Microsoft operating systems and the Java environment. It is less efficient for Western languages but more efficient for Eastern ones. A variant of UTF-16 is sometimes known as UCS-2.
- The **ISO-8859** series of encodings are 256-character ASCII supersets. They cannot support all of the Unicode characters; they can support only some particular language or family of languages. ISO-8859-1, also known as Latin-1, covers most Western European and African languages, but not Arabic. ISO-8859-2, also known as Latin-2, covers many Eastern European languages such as Hungarian and Polish.

If you want to be able to encode all Unicode characters, you probably want to use UTF-8. You will probably need to deal with the other encodings only when you are handed data in those encodings created by some other application.

7.6.1 `ord()` and `chr()`

Given a string representing one Unicode character, `ord()` returns an integer representing the Unicode code point of that character. For example, `ord('a')` returns the integer 97. Conversely, `chr(i)` returns the string representing a character whose Unicode code point is the integer *i*. For example, `chr(97)` returns the string 'a':

```
1 ord('a')
2 >>> 97
```

```
3 ord('7')
4 >>> 55
5 ord('%')
6 >>> 37
```

Conversely:

```
1 chr(97)
2 'a'
3 chr(55)
4 '7'
5 chr(37)
6 '%'
```

Chapter 8

Arrays

8.1 Lists as Arrays

A data structure is a way to organize data that we wish to process with a computer program. A one-dimensional array (or array) is a data structure that stores a sequence of (references to) objects. We refer to the objects within an array as its elements. The method that we use to refer to elements in an array is numbering and then indexing them. If we have n elements in the sequence, we think of them as being numbered from 0 to $n - 1$. Then, we can unambiguously specify one of them by referring to the i^{th} element for any integer i in this range.

Arrays consist of fixed-size data records that allow each element to be efficiently located based on its **findex**.

A two-dimensional array is an array of (references to) one-dimensional arrays. Whereas the elements of a one-dimensional array are indexed by a single integer, the elements of a two-dimensional array are indexed by a pair of integers: the first specifying a row, and the second specifying a column. In this chapter we will work with **lists as arrays** and refer to them as arrays thus setting the tone for next chapter. We will use arrays to perform various computations. The chapter will also serve as a review of most of the concepts learned so far and in fact the array descriptions are identical to that of lists first introduced in chapter 4.1.

Initializing Python arrays. The simplest way to create an array in Python is to place comma-separated literals between matching square brackets. For example, the code

```
1 SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
2 x = [0.30, 0.60, 0.10]
3 y = [0.50, 0.10, 0.40]
```

creates an array `SUITS[]` with four strings, and creates arrays `x[]` and `y[]`, each with three floats.

It is useful to think of references of the elements in an array as stored contiguously, one after the other, in your computer's memory, as shown in figure 8.1 for the `SUITS[]` array defined above.

Zero-based indexing. We always refer to the first element of an array `a[]` as `a[0]`, the second as `a[1]`, and so forth. It might seem more natural to refer to the first element as `a[1]`, the second

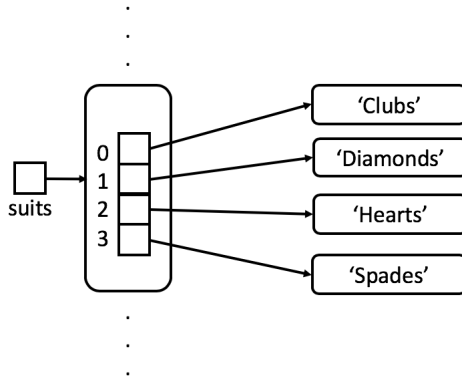


Figure 8.1: Array data structure

element as `a[2]`, and so forth, but starting the indexing with 0 has some advantages and has emerged as the convention used in most modern programming languages.

Array length. You can access the length of an array using Python’s built-in `len()` function: `len(a)` is the number of elements in `a[]`. In Python, we can use the `+=` operator to append elements to an array. For example, if `a[]` is the array `[1, 2, 3]`, then the statement `a += [4]` extends it to `[1, 2, 3, 4]`. More generally, we can make an array of `n` floats, with each element initialized to 0.0, with the code:

```

1 a = []
2 for i in range(n):
3     a += [0.0]
```

Bounds checking. You must be careful when programming with arrays. It is your responsibility to use legal indices when accessing an array element.

Mutability. An object is mutable if its value can change. Arrays are mutable objects because we can change their elements. For example, if we create an array `x = [.30, .60, .10]`, then the assignment statement `x[1] = .99` changes it to the array `[.30, .99, .10]`. The following code **reverses** the order of the elements in an array `a[]`:

```

1 n = len(a)
2 for i in range(n // 2):
3     temp = a[i]
4     a[i] = a[n-1-i]
5     a[n-1-i] = temp
```

Of course, since `a` is a list, you could simply use `a.reverse()` which modifies the original array in-place, i.e. no additional memory is required.

Iteration. The following code iterates over all elements of an array to compute the average of the floats that it contains:

```

1 total = 0.0
2 for i in range(len(a)):
3     total += a[i]
4 average = total / len(a)

```

Python also supports iterating over the elements in an array without referring to the indices explicitly. To do so, put the array name after the `in` keyword in a `for` statement, as follows:

```

1 total = 0.0
2 for v in a:
3     total += v
4 average = total / len(a)

```

Writing an array. You can write an array by passing it as an argument to `print()`. Each object in the array is converted to a string.

Aliasing. If `x[]` and `y[]` are arrays, the statement `x = y` causes `x` and `y` to reference the same array. This result has an effect that is perhaps unexpected, at first, because it is natural to think of `x` and `y` as references to two independent arrays. For example, after the assignment statements

```

1 x = [.30, .60, .10]
2 y = x
3 x[1] = .99

```

`y[1]` is also `.99`, even though the code does not refer directly to `y[1]`. This situation whereby two variables refer to the same object is known as **aliasing**.

Copying and slicing. So how do we make a copy `y[]` of a given array `x[]`? One answer to this question is to iterate through `x[]` to build `y[]`, as in the following code:

```

1 y = []
2 for v in x:
3     y += [v]

```

Copying an array is such a useful operation that Python provides language support for a more general operation known as **slicing**. So, the expression `a[i:j]` evaluates to a new array whose elements are `a[i], ..., a[j-1]`. Moreover, the default value for `i` is 0 and the default value for `j` is `len(a)`, so `y = x[:]` is equivalent to the code given above.

Example: Dot Product. Given two vectors of the same length, their **dot product** is the sum of the products of their corresponding components. If we represent the two vectors as one-dimensional arrays `x[]` and `y[]` that are each of length `n`, their dot product is easy to compute:

```

1 dotp = 0.0
2 for i in range(n):
3     dotp += x[i]*y[i]

```

Example: Representing playing cards. Suppose that we want to compose a program that processes playing cards. We might start with the following code:

```

1 SUITS = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
2 RANKS = ['2', '3', '4', '5', '6', '7', '8', '9', '10',
3         'Jack', 'Queen', 'King', 'Ace']

```

For example, we might use these two arrays to write a random card name, as follows:

```

1 rank = random.randrange(0, len(RANKS))
2 suit = random.randrange(0, len(SUITS))
3 print((RANKS[rank] + ' of ' + SUITS[suit]))

```

We might use the following code to initialize an array of length 52 that represents a deck of playing cards, using the two arrays just defined:

```

1 deck = []
2 for rank in RANKS:
3     for suit in SUITS:
4         card = rank + ' of ' + suit
5         deck += [card]
6 print(deck)

```

Frequently, we wish to **exchange** two elements in an array. Continuing our example with playing cards, the following code exchanges the cards at indices *i* and *j*:

```

1 temp = deck[i]
2 deck[i] = deck[j]
3 deck[j] = temp

```

Finally, the following code **shuffles** our deck of cards:

```

1 n = len(deck)
2 for i in range(n):
3     r = random.randrange(i, n)
4     temp = deck[r]
5     deck[r] = deck[i]
6     deck[i] = temp
7 print(deck)

```

8.2 2D Arrays

In many applications, a convenient way to store information is to use a table of numbers organized in a rectangular table and refer to rows and columns in the table. The mathematical abstraction corresponding to such tables is a **matrix**; the corresponding data structure is a **two-dimensional array**.

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}$$

Here, the numbers are arranged in rows and columns, where m and n represent the total number of rows and columns respectively. So, an example of a *square* 2x2 matrix is

$$A = \begin{bmatrix} 1 & 5 \\ -9 & 2 \end{bmatrix}$$

while a 3x4 matrix is

$$B = \begin{bmatrix} 4 & -3 & 11 & -13 \\ 1 & 0 & 7 & 20 \\ -12 & 2 & 5 & 6 \end{bmatrix}$$

Traditionally, we use the letter i to represent the row number and the letter j to represent the column number. So, if $i = 2$ and $j = 3$ then $B(i, j) = 7$. In Python, however, we need to keep in mind that we use zero-based indexing.

Initialization. The simplest way to create a two-dimensional array is to place comma-separated one-dimensional arrays between matching square brackets. For example, this matrix of integers having two rows and three columns

```
18 19 20
21 22 23
```

could be represented in Python using this array of arrays:

```
1 a = [[18, 19, 20], [21, 22, 23]]
```

Notice the following:

```
1 print(len(a)) # 2 : The array a is basically a "list" with two lists
2 print(len(a[0])) # 3 : The first row is a "list" has 3 elements
3 print(len(a[1])) # 3 : The second row has 3 elements
```

More generally, Python represents an m-by-n array as an array that contains m objects, each of which is an array that contains n objects. For example, this Python code creates an m-by-n array `a[][]` of floats, with all elements initialized to 0.0:

```
1 m=2
2 n=3
3 a = []
4 for i in range(m):
5     row = [0.0] * n
6     a += [row]
```

For convenience we can make this into a function that we can use later on:

```

1 def initMat(m, n):
2     mat = []
3     for i in range(m):
4         row = [0.0] * n
5         mat += [row]
6     return mat

```

Indexing. When `a` is a two-dimensional array, the syntax `a[i]` denotes a reference to its i^{th} row. The syntax `a[i][j]` refers to the object at row `i` and column `j`. To access each of the elements in a two-dimensional array, we use two nested for loops. For example, this code writes each object of the 3-by-4 array `B` given above, one row per line.

```

1 B = [[4, -3, 11, -13], [1, 0, 7, 20], [-12, 2, 5, 6]]
2 for i in range(len(B)):
3     for j in range(len(B[0])):
4         print(B[i][j], end=' ')
5     print()
6
7 # will print:
8 4 -3 11 -13
9 1 0 7 20
10 -12 2 5 6

```

Matrix operations. Typical applications in science and engineering involve representing matrices as two-dimensional arrays and then implementing various mathematical operations with matrix operands. For example, we can **add** two matrices `a` and `b` as follows:

Adding matrices. You can only add and subtract matrices of the same dimensions (size and shape), which means that you can add or subtract only the *corresponding elements*. Here is an example of how to add two 2x2 matrices:

```

1 def addMatrices(a,b):
2     '''adds two 2x2 matrices together'''
3     c = [[a[0][0]+b[0][0],a[0][1]+b[0][1]],
4         [a[1][0]+b[1][0],a[1][1]+b[1][1]]]
5     return c # note c is an array!
6
7 a = [[2,3],[5,-8]]
8 b = [[1,-4],[8,-6]]
9 c = addMatrices(a,b)
10 print(c)
11
12 [[3, -1], [13, -14]]

```

An even better way is to use loops and then we can generalize to add 2 nxn matrices:

```

1 def addMatrices(a,b):
2     '''adds two nxn matrices together'''

```

```

3     c = initMat(len(a), len(a[0])) # note this initialization
4     for i in range(len(a)): # loop over rows
5         for j in range(len(a[0])): # loop over columns
6             c[i][j] = a[i][j] + b[i][j]
7     return c
8
9 a = [[2,3],[5,-8]]
10 b = [[1,-4],[8,-6]]
11 print(addMatrices(a,b))
12
13 [[3, -1], [13, -14]]

```

Note the use of `initMat()`.

Multiplying matrices. Similarly, we can **multiply** two matrices $\vec{a}\vec{b}$ keeping in mind that $\vec{a}\vec{b}$ is defined if and only if the number of columns of \vec{a} equals the number of rows of \vec{b} . Thus, each element `c[i][j]` in the product of `a[i][j]` and `b[i][j]` is computed by taking the dot product of row `i` of `a[i][j]` with column `j` of `b[i][j]`. For example, a general formula to multiply two 2x2 matrices is:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & bf + bh \\ ce + dg & ch + dh \end{bmatrix} \quad (8.1)$$

This can be generalized to multiplication of $(n \times m) \times (m \times p)$. This may seem complicated but once we have the matrices as input we can figure out the number of rows and columns and implement an algorithm to carry out the multiplication.

We can implement matrix multiplication as follows:

```

1 def multMatrices(a ,b):
2     c = initMat(len(a), len(b[0])) # initialize 2D matrix
3     for i in range(len(a)): # iterate over 2 rows of a
4         for j in range(len(b[0])): # iterate over 2 columns of b
5             for k in range(len(b)): # iterate over rows/columns of b
6                 c[i][j] += a[i][k] * b[k][j] # dot product
7     return c
8
9 a = [[1, 2, 3], [4, 5, 6]] # 2x3 matrix
10 b = [[1, 2], [3, 4], [5, 6]] # 3x2 matrix
11 for r in multMatrices(a ,b):
12     print(r)
13
14 [22, 28]
15 [49, 64]

```

Although this is a straightforward implementation of matrix multiplication you should understand how it works. This implementation is simple but inefficient, especially for large arrays like the ones used in science and engineering. We will introduce more efficient ways to deal with numerical array computations in the next chapter.

Order Matters in Matrix Multiplication. An important fact about multiplying matrices is that AxB doesn't necessarily equal BxA , i.e. matrix multiplication is non-commutative. Try it with the example above.

Transforming Matrices. Multiplying matrices lets us transform them. For example, given a vector v one can *rotate* v by multiplying it with a rotation matrix R and obtain a new, rotated, vector v' :

$$v' = Rv \tag{8.2}$$

where R is the rotation matrix¹ in 2D:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Note that for the given R the dimensions of v (and v') must be 2×1 . Now, if $\theta = 90^\circ$ then R is going to rotate v in the **counterclockwise** direction and R will be given by

$$R = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

For example, if we start with a unit vector lying on the x-axis and rotate it by 90° then we'll end up with a unit vector lying along the y-axis:

$$\begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix}$$

And in Python this becomes:

```

1 R = [[0, -1], [1, 0]] # 90deg counter-clockwise Rotation matrix
2 v = [[1], [0]] # 2x1 vector, lying along x-axis
3 vp = multMatrices(R ,v) # rotate 90 degrees
4 print(vp)
5
6 [[0.0], [1.0]] # rotated 2x1 vector, now lying along y-axis

```

Of course, we can also rotate another matrix, say A , but that is a bit more difficult to visualize than with vectors.

Transposing matrices. An important concept in matrices is **transposition**, where the columns become the rows, and vice versa. In our example, we want to change some matrix B into B^T , the notation for "the B-matrix, transposed.". So, if

$$B = \begin{bmatrix} 4 & -3 & 11 & -13 \\ 1 & 0 & 7 & 20 \\ -12 & 2 & 5 & 6 \end{bmatrix}$$

then

$$B^T = \begin{bmatrix} 4 & 1 & -12 \\ -3 & 0 & 2 \\ 11 & 7 & 5 \\ -13 & 20 & 6 \end{bmatrix}$$

¹https://www.wikiwand.com/en/Rotation_matrix

We will be returning to all the matrix operations and concepts discussed in this chapter. However, we will be using a powerful Python package called NumPy.

8.3 Other Arrays in Python

This section is optional.

8.3.1 Basic Typed Arrays

Python's **array** module provides space-efficient storage of basic C-style data types like bytes, 32-bit integers, floating point numbers, and so on. We will not be using them in this class but it is worth mentioning them.

Arrays created with the **array** module are mutable and behave similarly to lists - except they are "typed arrays" constrained to a single data type.

Because of this constraint **array** objects with many elements are more space-efficient than lists (and tuples). The elements stored in them are tightly packed and this can be useful if you need to store many elements of the same type.

Also, **array** arrays support many of the same methods as regular lists. For example, to append to an array in Python you can just use the familiar `array.append()` method.

As a result of this similarity between Python lists and array objects, you might be able to use it as a "drop-in replacement" without requiring major changes to your application.

```
1 import array
2 arr = array.array('f', (1.0, 1.5, 2.0, 2.5))
3 arr[1]
4 1.5
5
6 # Arrays are "typed":
7 arr[1] = 'hello'
8 TypeError: "must be real number, not str"
```

8.3.2 Immutable Arrays of Unicode Characters

Python 3.x uses **str** objects to store textual data as immutable sequences of **Unicode** characters. Practically speaking that means a **str** is an immutable array of characters. Oddly enough it's also a recursive data structure - each character in a string is a **str** object of length 1 itself.

String objects are space-efficient because they are tightly packed and specialize in a single data type. If you are storing Unicode text you should use them. Because strings are immutable in Python modifying a string requires creating a modified copy. The closest equivalent to a "mutable string" is storing individual characters inside a list.

```
1 arr = 'abcd'
2 arr[1]
3 'b'
4
```

```

5 arr
6 'abcd'
7
8 # Strings are immutable:
9 arr[1] = 'e'
10 TypeError: "'str' object does not support item assignment"
11
12 # Strings are recursive data structures:
13 type('abc')
14 "<class 'str'"
15 type('abc'[0])
16 "<class 'str'"

```

8.3.3 Immutable Arrays of Single Bytes

Bytes objects are immutable sequences of single bytes (integers in the range of $0 \leq x \leq 255$). Conceptually they are similar to **str** objects and you can also think of them as immutable arrays of bytes.

Like strings, bytes have their own literal syntax for creating objects and they are space-efficient. Bytes objects are immutable, but unlike strings there is a dedicated "mutable byte array" data type called **bytearray** that they can be unpacked into.

```

1 arr = bytes((0, 1, 2, 3))
2 arr[1]
3 1
4
5 # Bytes literals have their own syntax:
6 arr
7 b'\x00\x01\x02\x03'
8
9 # Only valid "bytes" are allowed:
10 bytes((0, 300))
11 ValueError: "bytes must be in range(0, 256)"
12
13 # Bytes are immutable:
14 arr[1] = 23
15 TypeError: "'bytes' object does not support item assignment"

```

8.3.4 Mutable Arrays of Single Bytes

The **bytearray** type is a mutable sequence of integers in the range $0 \leq x \leq 255$. They are closely related to bytes objects with the main difference being that **bytearrays** can be modified freely - you can overwrite elements, remove existing elements, or add new ones. The **bytearray** object will grow and shrink appropriately.

```

1 arr = bytearray((0, 1, 2, 3))
2 arr[1]

```

```
3 1
4
5 # Bytearrays are mutable:
6 arr[1] = 23
7 arr
8 bytearray(b'\x00\x17\x02\x03')
9
10 arr[1]
11 23
```

There are a number of built-in data structures you can choose from when it comes to implementing arrays in Python. For simple applications, that may be all that you need. However, for scientific computing the most popular choice is use the NumPy module which we will cover next.

Chapter 9

Numpy

An array is an indexed sequence of objects, all of which are of the same type. Earlier, we implemented arrays using the Python list data type: a list object is an indexed sequence of objects, not necessarily of the same type. Using Python lists to implement arrays incurs substantial overhead, both in terms of memory (because Python must associate type information with each element) and time (because Python must perform a type check when accessing an element). Moreover, it is the programmer's responsibility to enforce the "all elements of the same type" constraint.

Now, we describe an alternative way to represent arrays in Python using the ndarray ("n-dimensional array") data type in the standard NumPy library: a ndarray object is an indexed sequence of objects, all of which are of the the same type - and NumPy enforces the "all elements of the same type" constraint. We use the informal term NumPy array to mean "an object of type ndarray."

Typically the elements of a NumPy array are numbers, such as floats or integers. As a result, there is minimal overhead in terms of memory (because NumPy need only associate type information with the array and not each element). Also, this representation can dramatically speed up certain types of "vectorized" computations because the array elements are stored contiguously in memory.

But there is more to NumPy than numeric arrays: the NumPy library also supports arrays whose elements are booleans and strings, and arrays whose elements are of data types that you define. There also is more to NumPy than arrays: the NumPy library provides many functions that work on "scalar" numeric objects. But the most valuable aspect of NumPy is its ability to manipulate numeric arrays; this chapter describes only that aspect.

9.1 When to Use Numpy

When should you use NumPy? One short answer is "when you need the functionality that NumPy provides." Indeed, as described later, NumPy provides a rich set of numeric array-handling functions and methods. Generally you should use the pre-defined (and thoroughly tested) functions and methods provided by NumPy instead of defining your own equivalent functions or methods.

Another short answer is "when you need your program to run faster." **The NumPy library was not written in Python**; instead it was written using the C programming language. Programs written in C run more quickly than those written in Python. So a Python program that implements arrays as NumPy arrays (maybe) will run faster than an equivalent Python program that implements arrays as ordinary Python lists.

However, there is more to the story. Since NumPy was written in C, there is a boundary between NumPy code and ordinary Python code. Crossing that boundary is expensive. That is, calling a NumPy function or method from ordinary Python code consumes more time than does calling an ordinary Python function or method. Similarly, returning a value from a NumPy function or method to ordinary Python code consumes more time than does returning a value from an ordinary Python function or method. With that in mind, suppose:

- Program A calls NumPy functions/methods many times, and each function/method call involves little computation.
- Program B calls NumPy functions/methods few times, and each function/method call involves much computation.

In that case Program B probably will benefit from the use of NumPy, but Program A probably will not.

9.2 Numpy Data Types

Since the NumPy library was written in the C programming language, its fundamental data types are, for the most part, those of C. Table 9.1 (taken from the online NumPy documentation) lists the NumPy fundamental data types: When you create a NumPy array, you specify the type of the array's elements. Normally you specify the element data type as a Python data type: int, float, bool, or complex. The Python int data type maps to the NumPy int_ data type. So if you create a NumPy array with elements of data type int, then internally within NumPy its elements are of type int_. Similarly, the Python float data type maps to the NumPy float_ data type, the Python bool data type maps to the NumPy bool_ data type, and the Python complex data type maps to the NumPy complex_ data type.

Usually you need not be concerned about the distinction between Python data types and NumPy data types. When an object of a Python data type is sent "across the boundary" to NumPy, the object automatically is converted to the appropriate Numpy data type. Conversely, when NumPy sends an object of a NumPy data type back to Python, and when that object is used in a context that demands an object of a Python data type, the object automatically is converted to the appropriate Python data type. We will henceforth ignore the distinction between Python and NumPy data types.

However the distinction between Python and NumPy data types can be important in programs that manipulate large-magnitude integers, that is, integers whose absolute values are large. Whereas Python int objects have unlimited range, the range of NumPy int_ objects is limited. A NumPy int_ object has range -2147483648 to 2147483647 (that is -2^{31} to $2^{31} - 1$) on systems that store integers using 32 binary digits, and -9223372036854775808 to 9223372036854775807 (that is, -2^{63} to $2^{63} - 1$) on systems that store integers using 64 binary digits.

So in NumPy it is possible for an expression to evaluate to an integer that is outside of the range that NumPy can store. When such an *overflow* occurs, NumPy evaluates the expression to an integer that is mathematically incorrect. Beware when manipulating large-magnitude integers in NumPy.

Table 9.1: NumPy Data Types

Data	Type Description
bool_	boolean (True or False) stored using 8 bits
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float_	Shorthand for float64
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

9.3 NumPy Array Fundamentals

There are several ways to import NumPy. The standard approach is to use a simple import statement:

```
1 import numpy
```

However, for large amounts of calls to NumPy functions, it can become tedious to write `numpy.X` over and over again. Instead, it is common to import under the briefer name `np`:

```
1 import numpy as np
```

This statement will allow us to access NumPy objects using `np.X` instead of `numpy.X`. It is also possible to import NumPy directly into the current namespace so that we don't have to use dot notation at all, but rather simply call the functions as if they were built-in:

```
1 from numpy import *
```

Never do this. If you do you will remove some of the nice organization that modules provide as well as potentially create namespace collisions. In this class we will assume that **`import numpy as np`** has been used.

How do we create Numpy arrays? An array can be created from a list:

```

1 import numpy as np
2 a = np.array([18, 19, 20, 21], int)

```

creates a one-dimensional NumPy array containing integers 18, 19, 20, and 21, and this statement:

```

1 b = np.array([[18.5, 19.3], [20.1, 21.0], [23.7, 24.9]], float)

```

creates a two dimensional NumPy array of floats having three rows and two columns. If you omit the second argument to `np.array()`, then the function infers the desired element type by examining the types of the values provided in the first argument.

To convert a NumPy array to a Python list, call the `tolist()` method. For example the expression `a.tolist()` evaluates to `[18, 19, 20, 21]`, and `b.tolist()` evaluates to `[[18.5, 19.3], [20.1, 21.0], [23.7, 24.9]]`.

You can reference an element of a one-dimensional NumPy array, just as you can reference an elements of a Python list, by specifying an index enclosed within square brackets. For example `a[1]` evaluates to 19. To reference an element of a two-dimensional NumPy array, specify the indices within square brackets, separated by commas. For example `b[1, 0]` evaluates to 20.1. Note that the syntax for referencing an element of a NumPy two-dimensional array differs from the syntax for referencing an element of a list of lists. (Recall that you would use the expression `b[1][0]` if `b` referenced a Python list of lists.)

Iteration over NumPy arrays works as expected:

```

1 for element in a:
2     print(element)
3
4 for row in b:
5     for element in row:
6         print(element)

```

Slicing a NumPy array also is straightforward. For example `a[1:2]` evaluates to the NumPy array `[19, 20]`. However, *slicing a numpy array does not generate a copy of the array*. For example, this statement:

```

1 e = a[1:2]

```

causes `e` to reference a subarray of the array referenced by `a` that is not distinct from `a`. Assigning some value to `e[0]` would change both `e[0]` and `a[1]`. Accordingly, the expression `a[:]` does not create a copy of the NumPy array referenced by `a`. Instead, to make a copy of a NumPy array you must call the `copy()` method:

```

1 e = a.copy()

```

Arrays can be multidimensional. Unlike lists, different axes are accessed using commas inside bracket notation. Here is an example with a two-dimensional array (e.g., a matrix):

```

1 a = np.array([[1, 2, 3], [4, 5, 6]], float)
2 print(a)
3 array([[1., 2., 3.],
4        [4., 5., 6.]])

```

Array slicing works with multiple dimensions in the same way as usual, applying each slice specification as a filter to a specified dimension. Use of a single ":" in a dimension indicates the use of everything along that dimension:

```
1 a = np.array([[1, 2, 3], [4, 5, 6]], float)
2 a[1,:]
3 array([ 4., 5., 6.])
4
5 a[:,2]
6 array([ 3., 6.])
7
8 a[-1:,-2:]
9 array([[ 5., 6.]])
```

Array Properties The *shape* property of an array returns a tuple with the size of each array dimension:

```
1 a.shape(2,3)
```

The *dtype* property tells you what type of values are stored by the array:

```
1 a.dtype
2 dtype('float64')
```

When used with an array, the *len* function returns the length of the first axis:

```
1 len(a)
2 2
```

The *in* statement can be used to test if values are present in an array:

```
1 2 in a
2 True
3 0 in a
4 False
```

Arrays can be reshaped using tuples that specify new dimensions. In the following example, we turn a ten-element one-dimensional array into a two-dimensional one whose first axis has five elements and whose second axis has two elements:

```
1 a = np.array(range(10), float)
2 a
3 array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
4 a = a.reshape((5, 2))
5 a
6 array([[ 0., 1.],
7         [ 2., 3.],
8         [ 4., 5.],
9         [ 6., 7.],
10        [ 8., 9.]])
11 a.shape
12 (5, 2)
```

Notice that the reshape function creates a new array and does not itself modify the original array. Transposed versions of arrays can also be generated, which will create a new array with the final two axes switched:

```
1 a = np.array(range(6), float).reshape((2, 3))
2 a
3 array([[ 0.,  1.,  2.],
4        [ 3.,  4.,  5.]])
5 a.transpose()
6 array([[ 0.,  3.],
7        [ 1.,  4.],
8        [ 2.,  5.]])
```

One-dimensional versions of multi-dimensional arrays can be generated with flatten:

```
1 a = np.array([[1, 2, 3], [4, 5, 6]], float)
2 a
3 array([[ 1.,  2.,  3.],
4        [ 4.,  5.,  6.]])
5 a.flatten()
6 array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

Finally, the dimensionality of an array can be increased using the newaxis constant in bracket notation:

```
1 a = np.array([1, 2, 3], float)
2 a
3 array([1., 2., 3.])
4 a[:,np.newaxis] # Note that this is now a column vector
5 array([[ 1.],
6        [ 2.],
7        [ 3.]])
8 a[:,np.newaxis].shape
9 (3,1)
10 b[np.newaxis,:]
11 array([[ 1.,  2.,  3.]])
12 b[np.newaxis,:].shape
13 (1,3)
```

Notice here that in each case the new array has two dimensions; the one created by newaxis has a length of one. The newaxis approach is convenient for generating the proper dimensioned arrays for vector and matrix mathematics.

9.3.1 Other ways to create arrays

The arange function is similar to the range function but returns an array:

```
1 np.arange(5, dtype=float)
2 array([ 0.,  1.,  2.,  3.,  4.])
3 np.arange(1, 6, 2, dtype=int)
```

```
4 array([1, 3, 5])
```

The functions `zeros` and `ones` create new arrays of specified dimensions filled with these values. These are perhaps the most commonly used functions to create new arrays:

```
1 np.ones((2,3), dtype=float)
2 array([[ 1.,  1.,  1.],
3        [ 1.,  1.,  1.]])
4 np.zeros(7, dtype=int)
5 array([0, 0, 0, 0, 0, 0, 0])
```

There are also a number of functions for creating special matrices (2D arrays). To create an identity matrix of a given size,

```
1 np.identity(4, dtype=float)
2 array([[ 1.,  0.,  0.,  0.],
3        [ 0.,  1.,  0.,  0.],
4        [ 0.,  0.,  1.,  0.],
5        [ 0.,  0.,  0.,  1.]])
```

The `eye` function returns matrices with ones along the *k*th diagonal:

```
1 np.eye(4, k=1, dtype=float)
2 array([[ 0.,  1.,  0.,  0.],
3        [ 0.,  0.,  1.,  0.],
4        [ 0.,  0.,  0.,  1.],
5        [ 0.,  0.,  0.,  0.]])
```

9.4 NumPy Array Operations

9.4.1 Basic operations

Many functions exist for extracting whole-array properties. The items in an array can be summed or multiplied:

```
1 >>> a = np.array([2, 4, 3], float)
2 >>> a.sum()
3 9.0
4 >>> a.prod()
5 24.0
```

In this example, member functions of the arrays were used. Alternatively, standalone functions in the NumPy module can be accessed:

```
1 >>> np.sum(a)
2 9.0
3 >>> np.prod(a)
4 24.0
```

For most of the routines described below, both standalone and member functions are available. A number of routines enable computation of statistical quantities in array datasets, such as the mean (average), variance, and standard deviation:

```
1 >>> a = np.array([2, 1, 9], float)
2 >>> a.mean()
3 4.0
4 >>> a.var()
5 12.666666666666666
6 >>> a.std()
7 3.5590260840104371
```

It's also possible to find the minimum and maximum element values:

```
1 >>> a = np.array([2, 1, 9], float)
2 >>> a.min()
3 1.0
4 >>> a.max()
5 9.0
```

The argmin and argmax functions return the array indices of the minimum and maximum values:

```
1 >>> a = np.array([2, 1, 9], float)
2 >>> a.argmin()
3 1
4 >>> a.argmax()
5 2
```

For multidimensional arrays, each of the functions thus far described can take an optional argument axis that will perform an operation along only the specified axis, placing the results in a return array:

```
1 >>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
2 >>> a.mean(axis=0)
3 array([ 2.,  2.])
4 >>> a.mean(axis=1)
5 array([ 1.,  1.,  4.])
6 >>> a.min(axis=1)
7 array([ 0., -1.,  3.])
8 >>> a.max(axis=0)
9 array([ 3.,  5.])
```

Like lists, arrays can be sorted:

```
1 >>> a = np.array([6, 2, 5, -1, 0], float)
2 >>> sorted(a)
3 [-1.0, 0.0, 2.0, 5.0, 6.0]
4 >>> a.sort()
5 >>> a
6 array([-1.,  0.,  2.,  5.,  6.])
```

For multidimensional arrays, each of the functions thus far described can take an optional argument `axis` that will perform an operation along only the specified axis, placing the results in a return array:

```
1 >>> a = np.array([[0, 2], [3, -1], [3, 5]], float)
2 >>> a.mean(axis=0)
3 array([ 2., 2.])
4 >>> a.mean(axis=1)
5 array([ 1., 1., 4.])
6 >>> a.min(axis=1)
7 array([ 0., -1., 3.])
8 >>> a.max(axis=0)
9 array([ 3., 5.])
10 Like lists, arrays can be sorted:
11 >>> a = np.array([6, 2, 5, -1, 0], float)
12 >>> sorted(a)
13 [-1.0, 0.0, 2.0, 5.0, 6.0]
14 >>> a.sort()
15 >>> a
16 array([-1., 0., 2., 5., 6.])
```

9.4.2 Comparison operators and value testing

Boolean comparisons can be used to compare members elementwise on arrays of equal size. The return value is an array of Boolean True / False values:

```
1 >>> a = np.array([1, 3, 0], float)
2 >>> b = np.array([0, 3, 2], float)
3 >>> a > b
4 array([ True, False, False], dtype=bool)
```

The results of a Boolean comparison can be stored in an array:

```
1 >>> c = a > b
2 >>> c
3 array([ True, False, False], dtype=bool)
```

The any and all operators can be used to determine whether or not any or all elements of a Boolean array are true:

```
1 >>> c = np.array([ True, False, False], bool)
2 >>> any(c)
3 True
4 >>> all(c)
5 False
```

9.4.3 Array item selection and manipulation

We have already seen that, like lists, individual elements and slices of arrays can be selected using bracket notation. Unlike lists, however, arrays also permit selection using other arrays. That is,

we can use array selectors to filter for specific subsets of elements of other arrays. Boolean arrays can be used as array selectors:

```
1 >>> a = np.array([[6, 4], [5, 9]], float)
2 >>> a >= 6
3 array([[ True, False],
4        [False, True]], dtype=bool)
5 >>> a[a >= 6]
6 array([ 6., 9.] )
```

Notice that sending the Boolean array given by $a_i=6$ to the bracket selection for a , an array with only the True elements is returned. We could have also stored the selector array in a variable:

```
1 >>> a = np.array([[6, 4], [5, 9]], float)
2 >>> sel = (a >= 6)
3 >>> a[sel]
4 array([ 6., 9.] )
```

In addition to Boolean selection, it is possible to select using integer arrays. Here, the integer arrays contain the indices of the elements to be taken from an array. Consider the following one-dimensional example:

```
1 >>> a = np.array([2, 4, 6, 8], float)
2 >>> b = np.array([0, 0, 1, 3, 2, 1], int)
3 >>> a[b]
4 array([ 2., 2., 4., 8., 6., 4.] )
```

9.4.4 Vector and matrix mathematics

NumPy provides many functions for performing standard vector and matrix multiplication routines. To perform a dot product,

```
1 >>> a = np.array([1, 2, 3], float)
2 >>> b = np.array([0, 1, 1], float)
3 >>> np.dot(a, b)
4 5.0
```

The dot function also generalizes to matrix multiplication:

```
1 >>> a = np.array([[0, 1], [2, 3]], float)
2 >>> b = np.array([2, 3], float)
3 >>> c = np.array([[1, 1], [4, 0]], float)
4 >>> a
5 array([[ 0., 1.],
6        [ 2., 3.]])
7 >>> np.dot(b, a)
8 array([ 6., 11.] )
9 >>> np.dot(a, b)
10 array([ 3., 13.] )
11 >>> np.dot(a, c)
```

```

12 array([[ 4.,  0.],
13        [ 14.,  2.]])
14 >>> np.dot(c, a)
15 array([[ 2.,  4.],
16        [ 0.,  4.]])

```

It is also possible to generate inner, outer, and cross products of matrices and vectors. For vectors, note that the inner product is equivalent to the dot product:

```

1 >>> a = np.array([1, 4, 0], float)
2 >>> b = np.array([2, 2, 1], float)
3 >>> np.outer(a, b)
4 array([[ 2.,  2.,  1.],
5        [ 8.,  8.,  4.],
6        [ 0.,  0.,  0.]])
7 >>> np.inner(a, b)
8 10.0
9 >>> np.cross(a, b)
10 array([ 4., -1., -6.])

```

NumPy also comes with a number of built-in routines for linear algebra calculations. These can be found in the sub-module `linalg`. Among these are routines for dealing with matrices and their inverses. The determinant of a matrix can be found:

```

1 >>> a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)
2 >>> a
3 array([[ 4.,  2.,  0.],
4        [ 9.,  3.,  7.],
5        [ 1.,  2.,  1.]])
6 >>> np.linalg.det(a)
7 -53.999999999999993

```

The inverse of a matrix can be found:

```

1 >>> b = np.linalg.inv(a)
2 >>> b
3 array([[ 0.14814815,  0.07407407, -0.25925926],
4        [ 0.2037037 , -0.14814815,  0.51851852],
5        [-0.27777778,  0.11111111,  0.11111111]])
6 >>> np.dot(a, b)
7 array([[ 1.00000000e+00,  5.55111512e-17,  2.22044605e-16],
8        [ 0.00000000e+00,  1.00000000e+00,  5.55111512e-16],
9        [ 1.11022302e-16,  0.00000000e+00,  1.00000000e+00]])

```

There are many other linear algebra operations which are beyond the scope of this class.

9.4.5 Statistics

In addition to the mean, var, and std functions, NumPy supplies several other methods for returning statistical features of arrays. The median can be found:

```

1 >>> a = np.array([1, 4, 3, 8, 9, 2, 3], float)
2 >>> np.median(a)
3 3.0

```

The correlation coefficient for multiple variables observed at multiple instances can be found for arrays of the form `[[x1, x2, ...], [y1, y2, ...], [z1, z2, ...], ...]` where `x, y, z` are different observables and the numbers indicate the observation times:

```

1 >>> a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
2 >>> c = np.corrcoef(a)
3 >>> c
4 array([[ 1. , 0.72870505],
5 [ 0.72870505, 1. ]])

```

Here the return array `c[i,j]` gives the correlation coefficient for the `i`th and `j`th observables. Similarly, the covariance for data can be found:

```

1 >>> np.cov(a)
2 array([[ 0.91666667, 2.08333333],
3 [ 2.08333333, 8.91666667]])

```

9.4.6 Random Numbers

An important part of any simulation is the ability to draw random numbers. For this purpose, we use NumPy's built-in pseudorandom number generator routines in the sub-module `random`. The numbers are pseudo random in the sense that they are generated deterministically from a seed number, but are distributed in what has statistical similarities to random fashion¹.

The random number seed can be set:

```

1 np.random.seed(293423)

```

The seed is an integer value. Any program that starts with the same seed will generate exactly the same sequence of random numbers each time it is run. This can be useful for debugging purposes, but one does not need to specify the seed and in fact, when we perform multiple runs of the same simulation to be averaged together, we want each such trial to have a different sequence of random numbers. If this command is not run, NumPy automatically selects a random seed (based on the time) that is different every time a program is run. An array of random numbers in the half-open interval `[0.0, 1.0)` can be generated:

```

1 >>> np.random.rand(5)
2 array([ 0.40783762, 0.7550402 , 0.00919317, 0.01713451, 0.95299583])

```

The `rand` function can be used to generate two-dimensional random arrays, or the `resize` function could be employed here:

```

1 >>> np.random.rand(2,3)
2 array([[ 0.50431753, 0.48272463, 0.45811345],
3 [ 0.18209476, 0.48631022, 0.49590404]])
4 >>> np.random.rand(6).reshape((2,3))

```

¹NumPy uses a particular algorithm called the Mersenne Twister to generate pseudorandom numbers

```

5 array([[ 0.72915152,  0.59423848,  0.25644881],
6        [ 0.75965311,  0.52151819,  0.60084796]])

```

To generate a single random number in [0.0, 1.0),

```

1 >>> np.random.random()
2 0.70110427435769551

```

To generate random integers in the range [min, max) use randint(min, max):

```

1 >>> np.random.randint(5, 10)
2 9

```

In each of these examples, we drew random numbers from a uniform distribution. NumPy also includes generators for many other distributions, some of which may be introduced later.

9.4.7 Note about operations

In many cases the same array operation can be performed using an operator, a method call, or a function call. For example, both of these expressions compute the *memberwise* sum of two arrays:

```

1 a + b                # Using an operator
2 numpy.add(a, b)     # Using a function call

```

and both of these expressions compute the dot product of two arrays:

```

1 a.dot(b)            # Using a method call
2 numpy.dot(a, b)    # Using a function call

```

Generally we use an operator if it is available, we use a method call only if an operator is unavailable, and we use a function call only if neither an operator nor a method call is available.

9.5 Numpy and Text IO

Numpy provides two functions to read in text data: savetxt and loadtxt. In the following simple example, we define an array x and save it as a textfile with savetxt:

```

1 import numpy as np
2 x = np.array([[1, 2, 3],
3              [4, 5, 6],
4              [7, 8, 9]], np.int32)
5 np.savetxt("digits.dat", x)

```

The file "digits.dat" is a text file and its content looks like this:

```

1.0000000000000000e+00 2.0000000000000000e+00 3.0000000000000000e+00
4.0000000000000000e+00 5.0000000000000000e+00 6.0000000000000000e+00
7.0000000000000000e+00 8.0000000000000000e+00 9.0000000000000000e+00

```

It is also possible to print the array in a special format, like for example with three decimal places or as integers. For this purpose we assign a format string to the third parameter 'fmt'. We saw in our first example that the default delimiter is a blank. We can change this behavior by assigning a string to the parameter "delimiter".

```
1 np.savetxt("digits2.dat", x, fmt="%2.3f", delimiter=",")
```

The file "digits2.dat" becomes:

```
1.000,2.000,3.000
4.000,5.000,6.000
7.000,8.000,9.000
```

We will read in now the file "digits.dat" using loadtxt:

```
1 y = np.loadtxt("digits.dat")
2 print(y)
3 [[1.  2.  3.]
4  [4.  5.  6.]
5  [7.  8.  9.]]
```

To read "digits2.dat":

```
1 y = np.loadtxt("digits2.dat", delimiter=",")
2 print(y)
3 [[1.  2.  3.]
4  [4.  5.  6.]
5  [7.  8.  9.]]
6
7 print(type(y))
8 <class 'numpy.ndarray'>
```

Using the Numpy loadtxt reads data into numpy arrays which allows us to use the full Numpy functionality on the data.

Chapter 10

Basic Plotting

10.1 Matplotlib: Pylab

The standard package for curve plotting in Python is Matplotlib. First we exemplify Matplotlib using `matplotlib.pylab`, which enables a syntax very close to that of Matlab.

As a first example let us plot the curve $y = t^2e^{-t^2}$ for t values between 0 and 3. First we generate equally spaced coordinates for t , say 51 values (50 intervals). Then we compute the corresponding y values at these points, before we call the `plot(t,y)` command to make the curve plot.

```
1 from matplotlib.pylab import *
2 def f(t):
3     return t**2*exp(-t**2)
4 t = linspace(0, 3, 51)
5 y = zeros(len(t))
6 for i in arange(len(t)):
7     y[i] = f(t[i])
8 # or simply y = f(t), which yields faster and shorter code
9 plot(t, y)
10 show()
```

The from `"matplotlib.pylab import *"` command performs two commands in the background: `"from numpy import *"` and import of all Matplotlib commands. We have already mentioned that this is bad practice. However, doing so creates a namespace that resembles Matlab-style syntax.

One can generate an image plot in PNG or other image formats. The `savefig` function saves the plot to files in various image formats:

```
1 savefig('exp_plot.png') # produce PNG
```

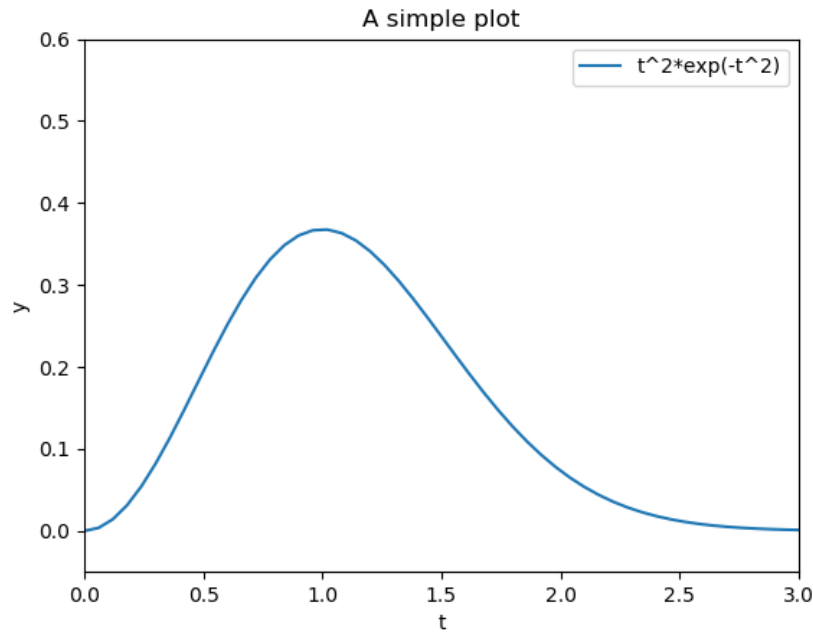
The x and y axes in curve plots should have labels, here t and y , respectively. Also, the curve should be identified with a label, or legend as it is often called. A title above the plot is also common. In addition, we may want to control the extent of the axes (although most plotting programs will automatically adjust the axes to the range of the data). All such things are easily added after the plot command:

```

1 xlabel('t')
2 ylabel('y')
3 legend(['t^2*exp(-t^2)'])
4 axis([0, 3, -0.05, 0.6]) # [tmin, tmax, ymin, ymax]
5 title('A simple plot')
6 show()

```

The generated image file is shown in figure 10.1.



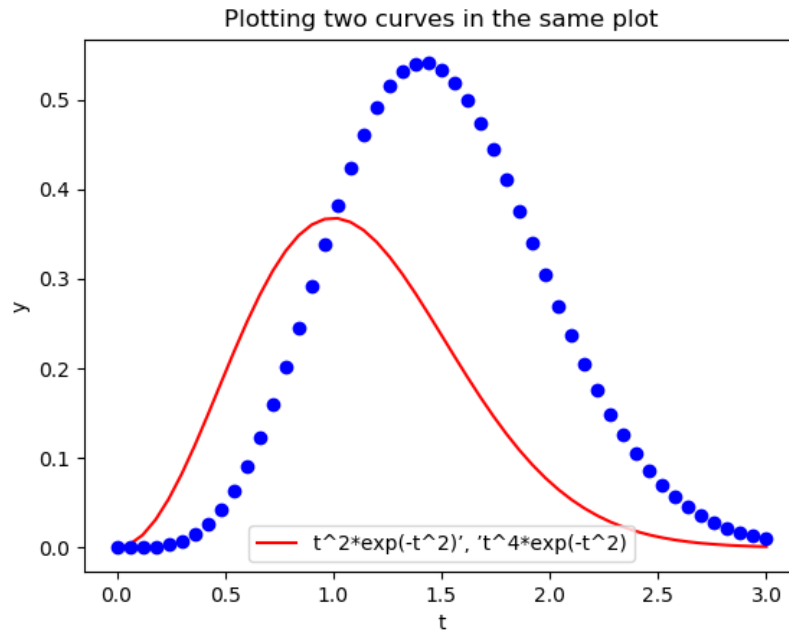
A common plotting task is to compare two or more curves, which requires multiple curves to be drawn in the same plot. Suppose we want to plot the two functions $f_1(t) = t^2e^{-t^2}$ and $f_2(t) = t^4e^{-t^2}$. We can then just issue two plot commands, one for each function:

```

1 def f1(t):
2     return t**2*exp(-t**2)
3 def f2(t):
4     return t**2*f1(t)
5 t = linspace(0, 3, 51)
6 y1 = f1(t)
7 y2 = f2(t)
8
9 plot(t, y1, 'r-')
10 plot(t, y2, 'bo')
11 xlabel('t')
12 ylabel('y')
13 legend(['t^2*exp(-t^2)?', ?t^4*exp(-t^2)'])
14 title('Plotting two curves in the same plot')
15 show()

```

In these plot commands, we have also specified the line type: r- means red (r) line (-), while bo means a blue (b) circle (o) at each data point. The legends for each curve is specified in a list where the sequence of strings correspond to the sequence of plot commands. The generated image file is shown in figure 10.1. We may also put plots together in a figure with r rows and c columns of

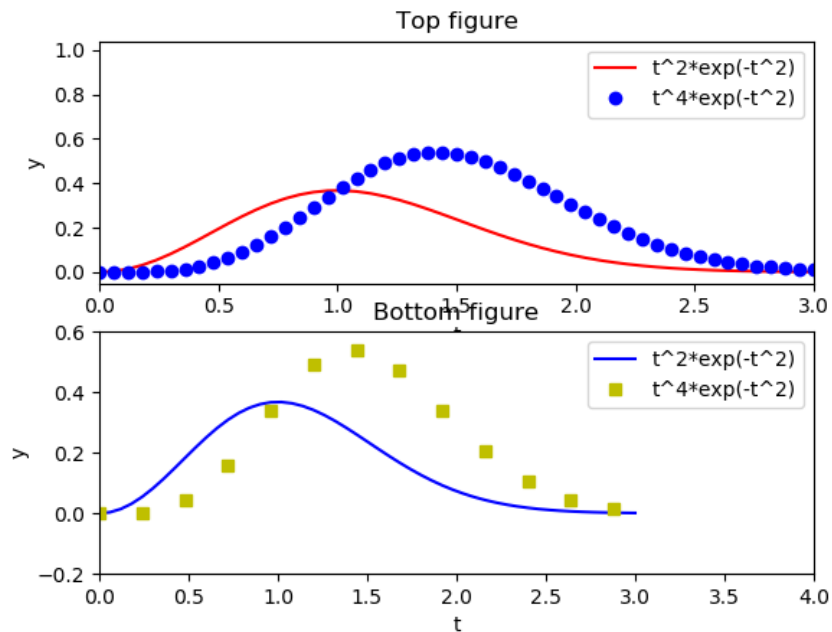


plots. The subplot(r,c,a) does this, where a is a row-wise counter for the individual plots.

```

1 subplot(2, 1, 1)
2 plot(t, y1, 'r-', t, y2, 'bo')
3 xlabel('t')
4 ylabel('y')
5 axis([t[0], t[-1], min(y2)-0.05, max(y2)+0.5])
6 legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
7 title('Top figure')
8
9 subplot(2, 1, 2)
10 t3 = t[::4]
11 y3 = f2(t3)
12 plot(t, y1, 'b-', t3, y3, 'ys')
13 xlabel('t')
14 ylabel('y')
15 axis([0, 4, -0.2, 0.6])
16 legend(['t^2*exp(-t^2)', 't^4*exp(-t^2)'])
17 title('Bottom figure')
18 show()

```

10.2 Matplotlib: Pyplot

In this class we do not promote the `matplotlib.pyplot` interface described in the previous section. However, you may use it for simple plots and quick prototyping. Instead, we recommend using the `matplotlib.pyplot` module. The standard practice is to prefix Numerical Python and Matplotlib functionality by short forms of their package names:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
```

The commands in `matplotlib.pyplot` are similar to those in `matplotlib.pyplot`. Most can typically be obtained by prefixing the pylab commands with `plt`:

```
1 plt.xlabel('t')
2 plt.ylabel('y')
3 plt.legend(['t^2*exp(-t^2)'])
4 plt.axis([0, 3, -0.05, 0.6]) # [tmin, tmax, ymin, ymax]
5 plt.title('A simple plot')
6 plt.show()
```

Once you have created a basic plot, there are numerous possibilities for fine-tuning the figure, i.e., adjusting tick marks on the axis, inserting text, etc. The Matplotlib website¹ is full of instructive examples on what you can do with this excellent package².

¹<https://matplotlib.org/>

²There are several other alternatives for specialized plotting. However, we will only use Matplotlib in this class.

Chapter 11

Object Oriented Programming

Python is an object-oriented language. It supports all the usual functionality of such languages such as classes, inheritance, and polymorphism. Object-oriented programming basically deals with creating modular, reusable bits of code, which are called objects. We have already been dealing with objects such as string objects and list objects¹. You are free to use Python without using any of its object oriented functionality by just sticking with functions and groups of statements as we have been doing throughout these notes. However, if you would like to create larger scale applications and systems, it may be more advantageous to use the object oriented programming paradigm.

11.1 Classes

A class packs a set of data (variables) together with a set of functions operating on the data. The goal is to achieve more modular code by grouping data and functions into manageable (often small) units. Most of the mathematical computations in this class can easily be coded without using classes, but in many problems, classes enable either more elegant solutions or code that is easier to extend at a later stage. In the non-mathematical world, where there are no mathematical concepts and associated algorithms to help structure the problem solving, software development can be very challenging. Classes may then improve the understanding of the problem and contribute to simplify the modeling of data and actions in programs. As a consequence, almost all large software systems being developed in the world today are heavily based on classes.

11.1.1 Defining a class

The syntax of defining class is as follows:

```
1 class class_name(parent_class_name):
2     <method_1_definition>
3     ...
4     <method_n_definition>
```

The class definition is divided into two parts: class header and class body.

¹In fact, everything in Python is an object.

The class header starts with class keyword followed by the name of the class, followed by the optional parent_class_name inside parentheses. The class_name and parent_class_name can be any valid identifier.

The parent_class_name class refers to the class you want to inherit from. This is known as **inheritance**. If you don't specify parent class name while defining a class it will be automatically set to **object**. We will discuss **inheritance** and **object** in more detail in the next sections.

In the next line, we have a class body, this is where we define methods to operate on data. Method definitions must be equally indented otherwise you will get a syntax error.

In the following example, we are defining a class to represent a Circle. The Circle class defines an attribute named radius and three methods namely `--init--()`, `get_area()` and `get_perimeter()` (see Circle.py).

```
1 import math
2
3 class Circle:
4
5     def __init__(self, radius):
6         self.radius = radius
7
8     def get_area(self):
9         return math.pi * self.radius ** 2
10
11    def get_perimeter(self):
12        return 2 * math.pi * self.radius
```

Let's step through the code line by line:

In line 1, we are importing math module because we will be using its pi constant in our methods.

In line 3, we have a class header, which starts with class keyword followed by the class name, which in this case is Circle, followed by a colon (:).

In the next line we have the class body, where we have defined the following three methods:

1. `--init--()` method.
2. `get_area()` method.
3. `get_perimeter()` method.

The syntax of defining methods is exactly the same as that of functions.

Notice that each method has a first parameter named **self**. In Python, the self parameter is required for every method. The self parameter refers to the object that invokes that method. Python uses self parameter to know which object to operate on inside the class. While calling the method you don't need to pass any value to self parameter, Python interpreter automatically binds the self parameter to the object when a method is called.

In lines 5-6, we have defined a method named `--init--`. The `--init--()` is a special method known as initializer or **constructor**. It is invoked everytime after a new object is created in memory. The purpose of the initializer method is to create the object's attribute with some initial value.

Apart from the `self` parameter, `__init__()` expects a `radius` parameter to provide an initial value to the `radius` attribute of the `Circle` object. The object's attribute is also commonly known as an **instance** variable. The methods that operate on instance variables are known as instance methods. Our `Circle` class has two instance methods `get_perimeter()` and `get_area()`. Each individual object has its own set of instance variables. These variables stores object's data. The scope of an instance variable and keyword `self` is limited to the body of the class. Inside a class, we use `self` to access object's attributes and methods. For example, we can use `self.var` to access instance variable named `var` and `self.foo()` to invoke the `foo()` instance method of an object.

Defining the constructor method (i.e `__init__()`) is not a requirement, If you don't define it then Python automatically supplies an empty `__init__()` method which does nothing.

In line 6, the instance variable `self.radius` is initialized to the value of the `radius` variable. That line of code creates an attribute named `radius` with an initial value for the object that was just created.

Unlike instance variable `radius` (on the left side), the `radius` variable on the right hand side of the assignment operator is a local variable and its scope is only limited to the `__init__()` method.

In lines 8 to 9, we have defined `get_area()` instance method which calculates and returns area of the circle.

In lines 11 to 12, we have defined `get_perimeter()` instance method which calculates and returns perimeter of the circle.

Notice that inside `get_area()` and `get_perimeter()` we are accessing instance variable `radius` using `self.radius` instead of just `radius`.

Now that we know how `Circle` class is defined, let's create some objects of the `Circle` class.

11.1.2 Creating objects

We can create objects from a class by calling class name as if it was a function:

```
1 ClassName()
```

However, if you have defined `__init__()` method then you would need to call class name with arguments as follows:

```
1 ClassName(arguments)
```

The arguments must match the parameters in the `__init__()` method without `self`. Otherwise, you will get an error.

Here is an example:

```
1 my_circle = Circle(5)
```

The above statement does the following things:

1. Creates an object of `Circle` class.
2. Invokes the `__init__()` method, passes this newly created `Circle` object to `self`, and the other argument (i.e 5) to the `radius` variable.
3. Creates an attribute named `radius` with an initial value for the object referenced by `self`. returns the `Circle` object

4. assigns the reference of the Circle object to the variable my_circle.

Note that my_circle only contains a reference (address) to the object, not the actual object.

11.1.3 Accessing Attributes and Methods

Once we have an object of a class, we can use it to access the object's attribute (or instance variable) and methods using the following syntax:

```
1 object.attribute # syntax to access attributes
2 object.method(arguments) # syntax to access instance methods
```

Here is how we can access attribute and methods of the Circle object (see circle_test1.py).

```
1 from circle import *
2
3 my_circle = Circle(5)
4
5 print("Circle of radius is",my_circle.radius)
6 print("Area of circle:", format(my_circle.get_area(), ".2f"))
7 print("Area of perimeter of circle:", format(my_circle.get_perimeter
      (), ".2f"), end="\n\n")
8
9 Circle of radius is 5
10 Area of circle: 78.54
11 Area of perimeter of circle: 31.42
```

Notice that while calling instance methods, we are not passing any value to the self parameter because Python automatically passes the reference to the object that was used to call the method to the self parameter. So in this case, the object referenced by the variable my_circle is passed to the self parameter. However, you are not allowed to pass an argument to self parameter in your code. If you try to do so, you will an error.

We can also change the object's attribute using the following syntax:

```
1 object.attribute = new_val
```

The following code changes the value of my_circle object's radius attribute from 5 to 10.

```
1 my_circle.radius = 10
```

Finally, we can create as many objects as we want. Each object will have its own set of attributes. Changing attributes for one object will not affect the attributes of other objects. For example (see circle_test2.py):

```
1 from circle import *
2
3 my_circle1 = Circle(5)
4 my_circle2 = Circle(10)
5 my_circle3 = Circle(15)
6
7 print("Address of Circle objects")
```

```

8 print("my_circle1:", id(my_circle1)) # print the address of Circle
   object referenced by variable my_circle1
9 print("my_circle2:", id(my_circle2)) # print the address of Circle
   object referenced by variable my_circle2
10 print("my_circle3:", id(my_circle3)) # print the address of Circle
   object referenced by variable my_circle3
11 print()
12
13 print("Address of radius attribute")
14 print("my_circle1:", id(my_circle1.radius)) # print the address of
   my_circle1's radius attribute
15 print("my_circle2:", id(my_circle2.radius)) # print the address of
   my_circle2's radius attribute
16 print("my_circle3:", id(my_circle3.radius), end="\n\n") # print the
   address of my_circle3's radius attribute
17
18 print("Initial value of radius attribute: ")
19 print("my_circle1's radius:", my_circle1.radius)
20 print("my_circle2's radius:", my_circle2.radius)
21 print("my_circle3's radius:", my_circle3.radius, end="\n\n")
22
23 # changing radius attribute of circle objects
24 my_circle1.radius = 50
25 my_circle2.radius = 100
26 my_circle3.radius = 150
27
28 print("After changing radius attribute of circle objects", end="\n\n")
   ")
29
30 print("Final value of radius attribute: ")
31 print("my_circle1's radius:", my_circle1.radius)
32 print("my_circle2's radius:", my_circle2.radius)
33 print("my_circle3's radius:", my_circle3.radius)
34
35 OUTPUT:
36
37 Address of Circle objects
38 my_circle1: 5236440
39 my_circle2: 5236608
40 my_circle3: 32036008
41
42 Address of radius attribute
43 my_circle1: 1586284752
44 my_circle2: 1586284912
45 my_circle3: 1586285072
46
47 Initial value of radius attribute:
48 my_circle1's radius: 5

```

```

49 my_circle2's radius: 10
50 my_circle3's radius: 15
51
52 After changing radius attribute of circle objects
53
54 Final value of radius attribute:
55 my_circle1's radius: 50
56 my_circle2's radius: 100
57 my_circle3's radius: 150

```

The `my_circle1`, `my_circle2` and `my_circle3` refers to three different `Circle` objects stored in distinct memory locations. In addition to that, each object's attributes are also stored in distinct memory locations.

11.1.4 Data Hiding

By default, the object's attributes are visible outside the class. This is the reason why we were able to access `radius` outside of `Circle`. Most of the time we don't give access to the object's attribute outside of the class because it may result in accidental corruption of the attribute's data. For example nothing is preventing us to store a string or list in `radius`:

```

1 my_circle = Circle(4)
2 my_circle.radius = "www"

```

We can prevent these problems by restricting access to the object's attribute outside of the class and by implementing **accessor** and **mutator** methods.

Data Hiding in Python can be achieved by defining private attributes. We can define a private attribute by starting its name with two underscore characters (`__`). So if we change `self.radius` to `self.__radius`, then we will not be able to access `radius` outside of the class. Similarly, we can define a private method by starting its name with two leading underscores (`__`). Private attributes and methods can only be accessed inside the class. If you try to access them outside of the class you will get an error.

To make the attribute value accessible outside the class we use accessor methods. An accessor method is simply a method that returns the value of an object's attribute but does not change it. They are also commonly known as getter methods or simply getters and they usually start with the word `get`. Here is the general format of the accessor method.

```

1 def get_attributeName(self):
2     return self.attributeName

```

Similarly, we can have mutator methods. A method which stores the value to object's attribute is known as mutator method. We call the mutator method when we need to change or set the value to the object's attribute. In addition to setting value to the object's attribute, they may also provide additional validation to validate the data before it is assigned to the object's attribute. Mutator method is also commonly known as setter methods or setters. Its general format is:

```

1 def set_attributeName(self, newvalue):
2     ## add data validation here
3     self.attributeName = newvalue

```

Here is a rewrite of Circle class which makes radius attribute private and also implements getter and setter method for the radius attribute.

```
1 import math
2
3 class Circle:
4
5     def __init__(self, radius):
6         self.__radius = radius #
7
8     def get_area(self):
9         return math.pi * self.__radius ** 2
10
11    def get_perimeter(self):
12        return 2 * math.pi * self.__radius
13
14    # getter method for radius attribute
15
16    def get_radius(self):
17        return self.__radius
18
19    # setter method for radius attribute
20
21    def set_radius(self, radius):
22        if not isinstance(radius, int):
23            print("Error: ", radius, "is not an int")
24            return
25        self.__radius = radius
```

Notice that the set_radius() method can only accept an integer argument, if you pass data of any other type, it will report an error. Data validation in set_radius() is achieved using *isinstance()* function. The *isinstance()* function is used to test the type of the given object. It's syntax is:

```
1 isinstance(object, class_name)
```

The object represent the object we want to test and class_name represent the name of the class. If object is an instance of class_name, then isinstance() returns True. Otherwise False.

11.1.5 Passing Objects as Arguments to Function

Just like built-in objects, we can pass objects of user-defined classes to a function or method.

The following program (see circle_test.py) shows how you can pass an object of type Circle to a function named print_circle_info().

```
1 c1 = Circle(5.4)
2 c2 = Circle(10.5)
3
4 def print_circle_info(circle_obj):
5     print("#####")
```



```

6     print("Radius of circle", format(circle_obj.get_radius(), "0.2f"
    ))
7     print("Perimeter of circle", format(circle_obj.get_perimeter(),
    "0.2f"))
8     print("Area of circle", format(circle_obj.get_area(), "0.2f"))
9     print("#####", end="\n\n")
10
11 print_circle_info(c1) # passing circle object c1 to
    print_circle_info()
12 print_circle_info(c2) # passing circle object c2 to
    print_circle_info()
13
14 OUTPUT:
15
16 #####
17 Radius of circle 5.40
18 Perimeter of circle 33.93
19 Area of circle 91.61
20 #####
21
22 #####
23 Radius of circle 10.50
24 Perimeter of circle 65.97
25 Area of circle 346.36
26 #####

```

11.2 Inheritance and Polymorphism

Inheritance is a mechanism which allows us to create a new class - known as **child class** - that is based upon an existing class - the **parent class**, by adding new attributes and methods on top of the existing class. When you do so, the child class inherits attributes and methods of the parent class.

Inheritance really shines when you want to create classes that are very similar. All you need to do is to write the code for the things that they have common in one class - the parent class. And then write code for things that are very specific in a different class - the child class. This saves you from duplicating a lot of code.

Suppose we are creating a program which deals with various shapes. Each shape has some common properties. For example, the color of the shape, whether it is filled or not and so on. In addition to that, there are some properties which vary from shape to shape. For example, area and perimeter. The area of the rectangle is width * length whereas the area of the circle is πr^2 . At first, it might be tempting to create classes for different shapes like this:

```

1 class Rectangle:
2     def __init__(self, color, filled, width, length):
3         self.__color = color
4         self.__filled = filled

```

```

5         self.__width = width
6         self.__length = length
7
8     def get_color(self):
9         return self.__color
10
11    def set_color(self, color):
12        return self.__color = color
13
14    def is_filled(self):
15        return self.__filled
16
17    def set_filled(self, filled):
18        return self.__filled
19
20    def get_area():
21        return self.__width * self.__length
22
23 class Circle:
24     def __init__(self, color, filled, radius):
25         self.__color = color
26         self.__filled = filled
27         self.__radius = radius
28
29     def get_color(self):
30         return self.__color
31
32     def set_color(self, color):
33         return self.__color = color
34
35     def is_filled(self):
36         return self.__filled
37
38     def set_filled(self, filled):
39         return self.__filled
40
41     def get_area(self):
42         return math.pi * self.__radius ** 2

```

Did you notice that amount of duplicate code we are writing?

Both classes share the same `__color` and `__filled` attribute as well as their getter and setter methods. To make the situation worse, If we want to update how any of these methods work, then we would have to visit each class one by one to make the necessary changes. By using inheritance, we can abstract out common properties to a general `Shape` class (parent class) and then we can create child classes such as `Rectangle`, `Triangle` and `Circle` that inherits from the `Shape` class. A child class inherits all the attributes and methods from its parent class, but it can also add attributes and methods of its own.

To create a child class based upon the parent class we use the following syntax:

```
1 class ParentClass:
2     # body of ParentClass
3     # method1
4     # method2
5
6 class ChildClass(ParentClass):
7     # body of ChildClass
8     # method 1
9     # method 2
```

In Object Oriented lingo, when a class c2 inherits from a class c1, we say class c2 extends class c1 or class c2 is derived from class c1.

The following program (see Shape.py) demonstrates inheritance in action. It creates a class named Shape, which contains attributes and methods common to all shapes, then it creates two child classes Rectangle and Triangle which contains attributes and methods specific to them only.

```
1 import math
2
3 class Shape:
4
5     def __init__(self, color='black', filled=False):
6         self.__color = color
7         self.__filled = filled
8
9     def get_color(self):
10        return self.__color
11
12    def set_color(self, color):
13        self.__color = color
14
15    def get_filled(self):
16        return self.__filled
17
18    def set_filled(self, filled):
19        self.__filled = filled
20
21
22 class Rectangle(Shape):
23
24    def __init__(self, length, breadth):
25        super().__init__()
26        self.__length = length
27        self.__breadth = breadth
28
29    def get_length(self):
30        return self.__length
31
```

```

32     def set_length(self, length):
33         self.__length = length
34
35     def get_breadth(self):
36         return self.__breadth
37
38     def set_breadth(self, breadth):
39         self.__breadth = breadth
40
41     def get_area(self):
42         return self.__length * self.__breadth
43
44     def get_perimeter(self):
45         return 2 * (self.__length + self.__breadth)
46
47
48 class Circle(Shape):
49     def __init__(self, radius):
50         super().__init__()
51         self.__radius = radius
52
53     def get_radius(self):
54         return self.__radius
55
56     def set_radius(self, radius):
57         self.__radius = radius
58
59     def get_area(self):
60         return math.pi * self.__radius ** 2
61
62     def get_perimeter(self):
63         return 2 * math.pi * self.__radius
64
65
66 r1 = Rectangle(10.5, 2.5)
67
68 print("Area of rectangle r1:", r1.get_area())
69 print("Perimeter of rectangle r1:", r1.get_perimeter())
70 print("Color of rectangle r1:", r1.get_color())
71 print("Is rectangle r1 filled ? ", r1.get_filled())
72 r1.set_filled(True)
73 print("Is rectangle r1 filled ? ", r1.get_filled())
74 r1.set_color("orange")
75 print("Color of rectangle r1:", r1.get_color())
76
77 c1 = Circle(12)
78
79 print("\nArea of circle c1:", format(c1.get_area(), "0.2f"))

```

```

80 print("Perimeter of circle c1:", format(c1.get_perimeter(), "0.2f"))
81 print("Color of circle c1:", c1.get_color())
82 print("Is circle c1 filled ? ", c1.get_filled())
83 c1.set_filled(True)
84 print("Is circle c1 filled ? ", c1.get_filled())
85 c1.set_color("blue")
86 print("Color of circle c1:", c1.get_color())
87
88 OUTPUT:
89
90 Area of rectagle r1: 26.25
91 Perimeter of rectagle r1: 26.0
92 Color of rectagle r1: black
93 Is rectagle r1 filled ? False
94 Is rectagle r1 filled ? True
95 Color of rectagle r1: orange
96
97 Area of circle c1: 452.39
98 Perimeter of circle c1: 75.40
99 Color of circle c1: black
100 Is circle c1 filled ? False
101 Is circle c1 filled ? True
102 Color of circle c1: blue

```

In lines 3-19, we have defined a Shape class. It is a parent class and only contains attributes and methods common to all shapes. This class defines two private attributes `__color` and `__filled`, then it provides getter and setter methods for those attributes.

In lines 22-45, we have defined a Rectangle class which inherits from Shape class. Pay close attention to the syntax we are using. This line tells us that Rectangle class extends the Shape class or Rectangle class is a child class of Shape class. Thus the Rectangle class inherits attributes and methods defined in the Shape class. In addition to that, the Rectangle class adds two private attributes, getter and setter methods for private attributes, as well as methods to calculate the area and perimeter of the rectangle.

Notice the code in line 25.

```

1 super().__init__()

```

In Python, we use `super()` function to call the parent class methods. So the above code calls Shape class's `__init__()` method. This is required to set the values of attributes in the parent class. Otherwise, when you try to access values of attributes defined in the parent class using getter or setter methods, you will get an error.

Similarly, In lines 48-63 we have defined a Circle class. Just like Rectangle, it extends the Shape class and adds few attributes and methods of its own.

The code in lines 66-86, creates Rectangle and Circle object and then calls `get_area()`, `get_perimeter()`, `get_filled()`, `get_color()`, `set_color()` and `set_filled()` methods on these objects one by one. Notice how we are able to call methods which are defined in the same class, as well as methods which are defined on the parent class.

11.3 Multiple Inheritance

Python allows us to derive a class from several classes at once, this is known as Multiple Inheritance. Its general format is:

```
1 Class ParentClass_1:
2     # body of ParentClass_1
3
4 Class ParentClass_2:
5     # body of ParentClass_2
6
7 Class ParentClass_3:
8     # body of ParentClass_1
9
10 Class ChildClass(ParentClass_1, ParentClass_2, ParentClass_3):
11     # body of ChildClass
```

The ChildClass is derived from three classes ParentClass_1, ParentClass_2, ParentClass_3. As a result, it will inherit attributes and methods from all the three classes.

The following program demonstrates multiple inheritance in action (see `multiple_inherit.py`):

```
1 class A:
2     def explore(self):
3         print("explore() method called")
4
5 class B:
6     def search(self):
7         print("search() method called")
8
9 class C:
10    def discover(self):
11        print("discover() method called")
12
13 class D(A, B, C):
14    def test(self):
15        print("test() method called")
16
17
18 d_obj = D()
19 d_obj.explore()
20 d_obj.search()
21 d_obj.discover()
22 d_obj.test()
23
24 OUTPUT:
25 explore() method called
26 search() method called
27 discover() method called
28 test() method called
```

11.4 Polymorphism and Method Overriding

In literal sense, **polymorphism** means the ability to take various forms. In Python, polymorphism allows us to define methods in the child class with the same name as defined in their parent class.

As we know, a child class inherits all the methods from the parent class. However, you will encounter situations where the method inherited from the parent class doesn't quite fit into the child class. In such cases, you will have to re-implement method in the child class. This process is known as **method overriding**.

If you have overridden a method in the child class, then the version of the method will be called based upon the type of the object used to call it. If a child class object is used to call an overridden method then the child class version of the method is called. On the other hand, if parent class object is used to call an overridden method, then the parent class version of the method is called.

The following program demonstrates method overriding in action (see polymorphism.py):

```
1 class A:
2     def explore(self):
3         print("explore() method from class A")
4
5 class B(A):
6     def explore(self):
7         print("explore() method from class B")
8
9
10 b_obj = B()
11 a_obj = A()
12
13 b_obj.explore()
14 a_obj.explore()
15
16 OUTPUT:
17 explore() method from class B
18 explore() method from class A
```

Here b_obj is an object of class B (child class), as a result, class B version of the explore() method is called. However, the variable a_obj is an object of class A (parent class), as a result, class A version of the explore() method is called.

If for some reason you still want to access the overridden method of the parent class in the child class, you can call it using the super() function as follows (see polymorphism2.py):

```
1 class A:
2     def explore(self):
3         print("explore() method from class A")
4
5 class B(A):
6     def explore(self):
7         super().explore() # calling the parent class explore()
8         print("explore() method from class B")
```

```

9
10
11 b_obj = B()
12 b_obj.explore()
13
14 OUTPUT:
15 explore() method from class A
16 explore() method from class B

```

11.5 Operator Overloading

Operator Overloading lets you redefine the meaning of operator respective to your class. It is the magic of operator overloading that we were able to use the + operator to add two numbers objects, as well as concatenate two string objects.

```

1 >>>
2 >>> 10 + 400
3 410
4 >>>
5 >>> "ten" + "sor"
6 'tensor'
7 >>>

```

Here + operator has two interpretations. When used with numbers it is interpreted as an addition operator whereas with strings it is interpreted as the concatenation operator. In other words, we can say that the + operator is overloaded for int class and str class.

So how do we redefine or overload an operator for a particular class?

Operator Overloading is achieved by defining a special method in the class definition. The names of these methods start and end with double underscores (_). The special method used to overload + operator is called `__add__()`. Both int class and str class implements `__add__()` method. The int class version of the `__add__()` method simply adds two numbers whereas the str class version concatenates the string.

If the expression is for the form `x + y`, Python interprets it as `x.__add__(y)`. The version of the `__add__()` method called depends upon the type of x and y. If x and y are int objects then int class version of the `__add__()` is called. On the other hand, if x and y are list objects then list class version of the `__add__()` method is called.

```

1 >>>
2 >>> x, y = 10, 20
3 >>>
4 >>> x + y
5 30
6 >>> x.__add__(y) # same as x + y
7 30
8 >>>
9 >>> x, y = [11, 22], [1000, 2000]
10 >>>

```



```

11 >>> x + y
12 [11, 22, 1000, 2000]
13 >>>
14 >>> x.__add__(y) # same as x + y
15 [11, 22, 1000, 2000]
16 >>>

```

The following program (see `op_overload.py`) demonstrates how we can override operators in a class.

```

1 import math
2
3 class Point:
4
5     def __init__(self, x=0, y=0):
6         self.__x = x
7         self.__y = y
8
9     # get the x coordinate
10    def get_x(self):
11        return self.__x
12
13    # set the x coordinate
14    def set_x(self, x):
15        self.__x = x
16
17    # get the y coordinate
18    def get_y(self):
19        return self.__y
20
21    # set the y coordinate
22    def set_y(self, y):
23        self.__y = y
24
25    # get the current position
26    def get_position(self):
27        return self.__x, self.__y
28
29    # change x and y coordinate by a and b
30    def move(self, a, b):
31        self.__x += a
32        self.__y += b
33
34    # overloading + operator
35    def __add__(self, point_obj):
36        return Point(self.__x + point_obj.__x, self.__y + point_obj.
37                       __y)
38
39    # overloading - operator
40    def __sub__(self, point_obj):

```

```

40         return Point(self.__x - point_obj.__x, self.__y - point_obj.
41             __y)
42
43     # overloading < operator
44     def __lt__(self, point_obj):
45         return math.sqrt(self.__x ** 2 + self.__y ** 2) < math.sqrt(
46             point_obj.__x ** 2 + point_obj.__y ** 2)
47
48     # overloading <= operator
49     def __le__(self, point_obj):
50         return math.sqrt(self.__x ** 2 + self.__y ** 2) <= math.sqrt
51             (point_obj.__x ** 2 + point_obj.__y ** 2)
52
53     # overloading > operator
54     def __gt__(self, point_obj):
55         return math.sqrt(self.__x ** 2 + self.__y ** 2) > math.sqrt(
56             point_obj.__x ** 2 + point_obj.__y ** 2)
57
58     # overloading >= operator
59     def __ge__(self, point_obj):
60         return math.sqrt(self.__x ** 2 + self.__y ** 2) >= math.sqrt
61             (point_obj.__x ** 2 + point_obj.__y ** 2)
62
63     ## overriding __str__ function
64     def __str__(self):
65         return "Point object is at: (" + str(self.__x) + ", " + str(
66             self.__y) + ")"
67
68 p1 = Point(4, 6)
69 p2 = Point(10, 6)
70
71 print("Is p1 < p2 ?", p1 < p2)    # p1 < p2 is equivalent to p1.
72     __lt__(p2)
73 print("Is p1 <= p2 ?", p1 <= p2) # p1 <= p2 is equivalent to p1.
74     __le__(p2)
75 print("Is p1 > p2 ?", p1 > p2)    # p1 > p2 is equivalent to p1.
76     __gt__(p2)
77 print("Is p1 >= p2 ?", p1 >= p2)  # p1 >= p2 is equivalent to p1.
78     __ge__(p2)
79 print("Is p1 == p2 ?", p1 == p2)   # p1 < p2 is equivalent to p1.
80     __eq__(p2)

```

```

76 p3 = p1 + p2 # p1 + p2 is equivalent to p1.__add__(p2)
77 p4 = p1 - p2 # p1 - p2 is equivalent to p1.__sub__(p2)
78
79 print() # print an empty line
80 print(p1) # print(p1) is equivalent to print(p1.__str__())
81 print(p2) # print(p2) is equivalent to print(p2.__str__())
82 print(p3) # print(p3) is equivalent to print(p3.__str__())
83 print(p4) # print(p4) is equivalent to print(p4.__str__())
84
85 OUTPUT:
86 Is p1 < p2 ? True
87 Is p1 <= p2 ? True
88 Is p1 > p2 ? False
89 Is p1 >= p2 ? False
90 Is p1 == p2 ? False
91
92 Point object is at: (4, 6)
93 Point object is at: (10, 6)
94 Point object is at: (14, 12)
95 Point object is at: (-6, 0)

```

The Point class defines two private attributes `__x` and `__y` which represent x and y coordinates in a plane. Then it defines getter and setter methods for those attributes. It also defines, `get_position()` and `move()` method to get the current position and change coordinates respectively.

In line 35, we have overloaded the `+` operator for the Point class. The `__add__()` method creates a new Point object by adding individual coordinates of one Point object to another Point object. Finally, it returns the newly created object to its caller. This allows us to write expressions like this:

```

1 p3 = p1 + p2

```

where `p1`, `p2` and `p3` are three Point objects.

Python interprets the above expression as `p3 = p1.__add__(p2)`, and calls the `__add__()` method to add two Point objects. The return value from the `__add__()` method is then assigned to `p3`. It is important to note that when the `__add__()` method is called, the value of `p1` is assigned to the self parameter and the value of `p2` is assigned to the `point_obj` parameter. The rest of the special methods works in a similar fashion.

11.6 object : The Base Class (Optional)

In Python, all classes inherit from the object class implicitly. It means that the following two class definitions are equivalent.

```

1 class MyClass:
2     pass
3
4 class MyClass(object):
5     pass

```

It turns out that the object class provides some special methods with two leading and trailing underscores which are inherited by all the classes. Here are some important methods provided by the object class.

1. `__new__()`
2. `__init__()`
3. `__str__()`

The `__new__()` method creates the object. After creating the object it calls the `__init__()` method to initialize attributes of the object. Finally, it returns the newly created object to the calling program. Normally, we don't override `__new__()` method, however, if you want to significantly change the way an object is created, you should definitely override it.

The `__str__()` method is used to return a nicely formatted string representation of the object. The object class version of `__str__()` method returns a string containing the name of the class and its memory address in hexadecimal. For example:

```
1 class Jester:
2     def laugh(self):
3         return print("laugh() called")
4
5 obj = Jester()
6 print(obj)
7
8 <__main__.Jester object at 0x0000000002167E80>
```

Sure, it is not very helpful. We can easily override this method by defining a method named `__str__()` in the Jester class as follows.

```
1 class Jester:
2     def laugh(self):
3         return "laugh() called"
4
5     def __str__(self):
6         return "A more helpful description"
7
8 obj = Jester()
9 print(obj)
10
11 A more helpful description
```

Appendix A

Using the Command Line

A.1 Using the Command Line

In CDS 230, you will mainly run Python programs two ways: using the Spyder IDE or using the command line. This is a small guide in using the command line.

The [command-line shell](#), sometimes called the command prompt or the terminal, is a tool that lets you control your computer using only textual commands. It offers a lot of power and simplicity (simplicity is different from ease of use).

Just like with a graphical file browser such as the Finder or Windows Explorer, there is a "current directory" that you are currently working in. ("Directory" and "folder" are synonyms.) You can issue commands that operate in that directory, or you can change the current directory.

This guide presents an example transcript of using the shell for Unix (Mac/Linux) and Windows machines. The transcript assumes that the student has already installed the Anaconda Python Distribution, and has created the CDS-230 directory structure as described in the lecture. When you run similar commands, there may be slight differences from the example transcript, such as the number, names, and times of files.

See the section that is relevant to you:

A.1.1 Mac/Linux

Here are most of the commands you will need to use:

- *pwd* - print the absolute pathname of your current working directory
- *cd directory* - change your working directory to the given directory
- *cd ..* - change your working directory to the parent of the current working directory
- *ls* - list the contents of the current directory ("ls" is short for "list")
- *mkdir cds-230* - *mkdir* creates a directory named cds-230
- *python* - run the Python interpreter

- `python program.py` - run the Python program that is stored in the `program.py` file You can open a command-line shell by running the `terminal` program.

In the example below, `$` is the prompt at which the user types commands. What follows the `$` prompt was printed by the command-line shell.

```
$ pwd
/home/me
$ ls
Desktop    Downloads  Music    Pictures    Public    Templates  Videos
Documents  Dropbox    Old      Programming Software  Ubuntu One  VirtualBox VMs
$ cd Desktop
$ pwd
/home/me/Desktop
$ ls
cds-230
$ cd cds-230
$ pwd
/home/me/Desktop/cds-230
$ ls
data scripts
$ cd scripts
$ pwd
/home/me/Desktop/cds-230/scripts
$ ls
helloworld.py
$ python helloworld.py
Hello world!
```

A.1.2 Windows

Here are most of the commands you will need to use:

- `echo %cd%` - print the absolute pathname of your current working directory
- `cd directory` - change your working directory to the given directory
- `cd ..` - change your working directory to the parent of the current working directory
- `dir` - list the contents of the current directory ("`ls`" is short for "list")
- `mkdir cds-230` - `mkdir` creates a directory named `cds-230`
- `python` - run the Python interpreter
- `python program.py` - run the Python program that is stored in the `program.py` file You can open a command-line shell by running the `terminal` program.

You can open a command-line shell by running the cmd program. You should have a Command Prompt shortcut located in the Start Menu, in the Accessories submenu of All Programs, or on the Apps screen for Windows 8. About.com has [more detailed instructions about starting the command prompt](#).

In the example below, `C:\Users\Me>` is the prompt at which the user types commands. What follows the prompt was printed by the command-line shell.

```
C:\Users\Me>echo %cd%
C:\Users\Me
```

```
C:\Users\Me>dir
Directory of C:\Users\Me

06/02/2012  08:11 PM    <DIR>          .
06/02/2012  08:11 PM    <DIR>          ..
07/18/2012  05:03 PM    <DIR>          Contacts
01/10/2013  07:24 PM    <DIR>          Desktop
07/18/2012  05:03 PM    <DIR>          Documents
01/09/2013  09:59 PM    <DIR>          Downloads
07/18/2012  05:03 PM    <DIR>          Favorites
07/18/2012  05:03 PM    <DIR>          Links
07/18/2012  05:03 PM    <DIR>          Music
11/28/2012  09:19 PM    <DIR>          Pictures
11/29/2012  01:42 AM    <DIR>          Saved Games
07/18/2012  05:03 PM    <DIR>          Searches
11/27/2012  09:06 PM    <DIR>          Videos
```

```
C:\Users\Me>cd Desktop
```

```
C:\Users\Me\Desktop>mkdir cds-230
C:\Users\Me\Desktop>dir
Directory of C:\Users\Me\Desktop
```

```
01/10/2013  07:25 PM    <DIR>          .
01/10/2013  07:25 PM    <DIR>          ..
01/10/2013  07:25 PM    <DIR>          cds-230
```

```
C:\Users\Me\Desktop>cd cds-230
```

```
C:\Users\Me\Desktop>mkdir scripts
C:\Users\Me\Desktop>mkdir data
C:\Users\Me\Desktop\cds-230>dir
Directory of C:\Users\Me\Desktop\cds-230
```

```
01/10/2013  07:25 PM    <DIR>          .
01/10/2013  07:25 PM    <DIR>          ..
01/10/2013  07:25 PM    <DIR>          data
```

```
01/10/2013 07:24 PM <DIR>          scripts
```

```
C:\Users\Me\Desktop\cds-230>cd scripts
```

```
C:\Users\Me\Desktop\cds-230\homework2>dir
```

```
Directory of C:\Users\Me\Desktop\cds-230\scripts
```

```
01/10/2013 07:24 PM <DIR>          .
```

```
01/10/2013 07:24 PM <DIR>          ..
```

```
01/09/2013 09:26 PM          13  helloworld.py
```

```
C:\Users\Me\Desktop\cds-230\scripts>python helloworld.py
```

```
Hello world!
```


Appendix B

Computational Problem Solving

Computational problem solving does not simply involve the act of computer programming. It is a process, with programming being only one of the steps. Before a program is written, a design for the program must be developed. And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested. These steps are outlined below.

ANALYSIS

Clearly understand the problem
Know what constitutes a solution

DESIGN

Determine what type of data is needed
Determine how the data is to be structured
Find another design appropriate algorithm

IMPLEMENTATION

Represent data within programming language
Implement algorithms in programming language

TESTING

Test the program on a selected set of problem instances
Correct and understand the causes of any errors found

1. ANALYSIS

- (a) Understanding the problem. Once a problem is clearly understood, the fundamental computational issues for solving it can be determined.
- (b) Knowing what constitutes a solution. For some problems, there is only one solution. For others, there may be a number (or infinite number) of solutions. Thus, a program may be stated as finding

- A solution
- An approximate solution
- A best solution
- All solutions

2. DESIGN

- (a) Describing the data needed. This, of course, depends on the problem at hand. We can use a list, a table, a matrix, etc.
- (b) Describing the Needed Algorithms. For some problems, there is only one solution. When solving a computational problem, either suitable existing algorithms may be found or new algorithms must be developed. Algorithms that work well in general but are not guaranteed to give the correct result for each specific problem are called *heuristic algorithms*.

3. IMPLEMENTATION Design decisions provide general details of the data representation and the algorithmic approaches for solving a problem. The details, however, do not specify which programming language to use, or how to implement the program. That is a decision for the implementation phase. Since we are programming in Python, the implementation needs to be expressed in a syntactically correct and appropriate way, using the instructions and features available in Python.

4. TESTING Software testing is a crucial part of software development. Testing is done incrementally as a program is being developed, when the program is complete, and when the program needs to be updated.

Appendix C

References

Tutorials

Tutorials for beginners:

https://www.w3schools.com/PYTHON/python_lists.asp

<https://www.tutorialspoint.com/python/>

A Python tutorial from the official Python website:

<https://docs.python.org/3/tutorial//>

For exact syntax and semantics of the Python language:

<http://docs.python.org/3/>

Reference manual of the standard library:

<http://devdocs.io/python>

Online Tools

The Python Online Tutor allows you to visualize execution of Python code.

<http://people.csail.mit.edu/pgbovine/python/tutor.html>

Online Python interpreter: Just in case Jupyter Notebook is not enough.

https://www.onlinegdb.com/online_python_interpreter

Modeling and Simulation

This book has material similar in spirit to CDS230 but slightly different approach:

<http://greenteapress.com/wp/modsimpy>