# **Beginning Programming with Python**

Dan MacLean

May 2022

# Table of contents

Motivation
------------

1	Wor	king with Data	6
	1.1	Using variables as names for data	7
	1.2	Python has three main types of function	7
		1.2.1 Built-In Functions and Operators	3
		1.2.2 Functions from packages	3
		1.2.3 Object Methods	9
	1.3	Objects and types in Python	0
		1.3.1 String Objects	0
		1.3.2 Number Objects	2
	1.4	Quiz	2
2	Data	a Structures 14	4
	2.1	Lists are ordered linear collections of items 14	4
		2.1.1 List use $\ldots \ldots \ldots$	5
		2.1.2 List methods $\ldots$	6
	2.2	Dictionaries are unordered collections 10	б
		2.2.1 Dictionary Methods	7
	2.3	Quiz	7
3	Mak	sing Choices and Controlling Program Flow 19	9
-	3.1	Conditionals check values and adjust which code is run	9
	3.2	Indentation in Python is Pretty Important	0
	3.3	More ifs	1
		3.3.1 Testing equality	2
	3.4	Logical operators in tests	2
		3.4.1 What is True and what is False	2
	3.5	That's not not what I meant	3
	3.6	Quiz	3
4	Rep	eating actions with Loops 2	5
	4.1	For Loops	6
		4.1.1 Choosing loop variable names	8
		4.1.2 Loop variable lifespan	9
		4.1.3 Looping over a list	9

5

		4.1.4 Looping over a range of numbers	30
		4.1.5 Looping over a dictionary	30
	4.2	While Loops	31
	4.3	Quiz	32
_			~ •
5	Use	r Functions	34
	5.1	Building your own function	34
	5.2	Composing functions	36
	5.3	Variables inside and outside Functions	36
	5.4	Designing programs to use functions	37
	5.5	Quiz	37
6	Wor	king with Python Packages	40
U	61	Using and Installing Packages	40
	6.2	The vefny Package	41
	0.2	6.2.1 VCF files - a brief introduction	41
		6.2.2 The worf Reader object	42
		6.2.3 The ver Becord object	42
		6.2.4 Object Attributes	40
	63	Ouiz	40
	0.5	Quiz	44
7	Put	ting code into scripts	45
	7.1	The script header	45
	7.2	The filename extension	46
	7.3	Running a python script from the command line	46
	7.4	Getting Options from the Command Line	46
	7.5	Quiz	47
		•	
8	Put	ting It All Together	48
	8.1	A Challenge in Python	48
Λ.	nnn	dicos	18
	phen		70
Α	Inst	alling what you need	49
	A.1	Prerequisites	49
	A.2	Installing Python 3 with Anaconda	49
		A.2.1 Note for macOS users	49
		A.2.2 Note for Windows users	49
		A.2.3 Note for Linux Users	50
	A.3	Starting a Jupyter Notebook	50
	-	A.3.1 macOS	50
		A.3.2 Windows	50
			50
		A.3.3 Linux	- ()U

В	Acknowledgements	52
	A.5 Installing a Text Editor	. 51
	A.4 Installing Python Packages with conda	. 51

# Motivation

Programming is a pretty weird skill. It boils down to shouting at a computer to make it do stuff for you. So why would you want to learn to do it? There are plenty of reasons, some good and some bad. Some good ones that apply to working in biology and bioinformatics are time-saving, turning your computer from a limited 'appliance' to a general 'power tool' for your research and because it's a skill that can help you develop a more precise, disciplined and abstract way of thinking.

The main obstacle that most people encounter when learning to program is the surprisingly wide range of concepts and implementations of concepts that you need to know in order to achieve something. This can make it intimidating and tedious for those starting out.

The aim of this course is to introduce you to just enough of these to enable you to do useful stuff with Python. The things you'll learn here will seem quite abstract and disconnected at first but hopefully by the end of the course you'll be able to string them together to make something useful - and understand what's going on.

In this course, we'll use Python 3 - a widely used very powerful but (all things considered) user-friendly language that suits beginners and experts alike.

We'll use the bare-bones of Python 3. Python is a very broad language with a lot of functionality, a lot of it in optional packages that you can install whenever you need them. We'll only touch the surface of what is possible - but what we learn will be foundation enough to build pretty much anything on.

By the end of this course you'll have seen and used enough Python to be able to go forward and be able to start to tackle any sort of programming challenge. You'll still need your problem solving skills, tenacity and determination to do tackle your future challenges, but at least you'll know Python.

Happy Programming!

# 1 Working with Data

#### 1. Questions:

- How do I deal with data in Python?
- 2. Objectives:
- Defining variables and using them in functions
- Using simple data objects: strings and numbers
- Using package functions and object methods
- 3. Keypoints:
- variables are handy names for data objects
- data objects are used by functions
- functions can be stored in packages
- methods available depend on the object we're talking about

In any Python program we will have some data and some objective to achieve - something to do with that data. Python provides many data types and many ways of working with that data.

Manipulating data is done with functions. Data is stored in objects. In this section we'll look at functions and objects and their interaction.

Let's see the simplest example of this workflow, let's take a string (a text carrying object data type) and use it in a function.

print("Hello, world!")

Hello, world!

In this example the string "Hello, World!", is being given to the print() function. Functions are bits of code that do stuff to data. The print() function just prints the data that you pass it to the screen. We pass data to functions by putting the data in the brackets after the function name.

## 1.1 Using variables as names for data

We don't usually use data directly. Instead we use a name that refers to a piece of data - a variable. Variables are just names that represent a bit of data. It's called a variable because the data the variable is associated with can change. We assign a name to data by using the assignment symbol the = sign. The data associated with a variable can be changed by re-assignment, allowing us to reuse the name.

We can use variables as if they are the data they point to

```
x = "Hello, world!"
print(x)
x = 100
print(x)
Hello, world!
100
```

And variables are independent of one another, actions on one don't affect another

```
weight_kg = 65
weight_pounds = 2.2 * weight_kg
print(weight_kg)
print(weight_pounds)
```

65 143.0

## **1.2** Python has three main types of function

Python is **full** of functions. So many in fact that coming up with names can be a problem! This is a serious issue as you have to refer to functions by name. To resolve this problem Python keeps its functions in different places in its library, and the way we call the function changes depending on where the function 'lives'. There are three basic function types:

• Built-In Functions and Operators

- Package Functions
- Object Methods

#### 1.2.1 Built-In Functions and Operators

Python has some functions that can be called directly from anywhere in a program. These are listed here https://docs.python.org/3.3/library/functions.html. We've already seen print() 'and there are some common ones we'll come across later. You can spot a built-in function because it has a single-word name followed by brackets.

Related to built-in functions are operators. The things you'll use most are the mathematical operators that work as you would expect from your knowledge of maths. So they include things like, +, -, \*, / etc.

```
print(1 + 1)
print(2 * 2)
print(3 - 3)
print(4 / 4)
2
4
0
1.0
```

Some operators change what they do depending on the things you ask them to operate on. For instance, we can add strings?!

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

#### HelloWorld!

This is supposed to be a way of making the language more intuitive and readable at the user end of things. Most times you see operators, they should be pretty obvious.

#### 1.2.2 Functions from packages

Another source of functions is external packages - extensions to Python for use in particular problem domains. We can load in a package using import. Let's import the random Python package that provides functions for generating random numbers.

import random

We can access the functions in this package by using the package name and the dot (.) syntax and the function name. Let's call the numpy function randrange() which gives us a random number between two limits.

```
number = random.randrange(1, 10)
print( number )
```

9

### 1.2.3 Object Methods

Lumps of data are represented in the computer in things called objects. An object is basically a bit of data with some functions attached. This means each piece of data comes with the code to manipulate it. These attached functions are called **methods**.

We access data's methods using the . syntax again, so this time you have variable\_name.method(), read this as you telling the object the variable points at to do method() to itself. This is simpler than it sounds.

Consider a variable x pointing to a string object. We might use it with .capitalize() as follows

```
x = "hello, world!"
print( x.capitalize() )
```

Hello, world!

This does mean that the methods are closely tied to the data. Look what happens when we try to use .capitalize() on a number

```
y = 100
print( y.capitalize() )
```

```
Error in py_run_string_impl(code, local, convert) :
   AttributeError: 'int' object has no attribute 'capitalize'
```

Python just throws an error. Basically this error is saying that int (integer, a number object) doesn't have a method called capitalize.

Loosely, methods are functions that only apply to particular object types.

We need to know what methods an object has before we can work on them. We can find this by reading the documentation for the object type. And Python will give us the type with the type() function

```
print( type(x) )
```

<class 'str'>

We can see that x contains a str - a string. The easiest place to find the Python documentation is online. Googling Python 3 str shows us this page https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str, which shows us all the String methods. This works well for finding all methods for objects of other types.

## 1.3 Objects and types in Python

Let's examine some object types.

Python knows many types of object, most things are an object of some type. Three common object types are:

- strings
- integer numbers
- floating point numbers

### 1.3.1 String Objects

The term 'string' is computer jargon for text data, usually a single lump of text data treated as a whole. To create a strings we simply have to add single or double quotes around some text, for example:

weight\_kg\_text = 'weight in kilograms'

Having actual numbers in there doesn't make a string a number type - the following is still a string - it just happens to be made up of number like characters. Let's see what happens if we try and treat it like a number.

```
phone_number = "01818118181"
print( phone_number * 2 )
```

#### 0181811818101818118181

Here the \* operator has modified itself to work on a string and repeated the string! Usually though the program will crash out, giving an operator the wrong data will confuse the program.

### 1.3.1.1 Indexing a string (Slicing)

We can access a single character in a string using indexing - basically asking for a character at a position. The syntax uses the square brackets.

```
print( phone_number[2] )
```

8

Note that using the index [2] gives us the *third* character - computer languages tend to count from 0.

We can get a longer subsection of a string using indexing as well - this is a technique that accesses a part of the data given a start and end point along the string.

```
dialling_code = phone_number[0:3]
print(dialling_code)
```

#### 018

We just use the square brackets to indicate the start and stop points of the slice we want to extract, literally [start:end]. The start here is 0 meaning the first character, the end here is 3, but it means up to but not including the end.

print( dialling\_code )

018

That's why we only get the first three characters from the string and not 4 ie 0,1,2,3. The way to remember this is that the length of the resulting slice is end - start.

There are many string operations and methods, you can see them in the documentation at https://docs.python.org/3/library/stdtypes.html#textseq

### 1.3.2 Number Objects

Numbers in Python come in two types, whole numbers (called integers) and numbers with a decimal part (called floating point numbers).

In the example above, variable weight\_kg has an integer value of 65. To create a variable with a floating point value, we can execute:

```
weight_kg = 65.0
```

The difference is important in some cases. You can convert type explicitly using the int() and float() functions.

```
print( float(1) + 3 )
print( int( 10.0 / 3.0 ) )
```

4.0 3

## 1.4 Quiz

1. What values do the variables mass and age have after each statement in the following program? Test your answers by executing the commands.

```
mass = 47.5
age = 122
mass = mass * 2.0
age = age - 20
```

2. What does the following program print out?

```
first, second = 'Grace', 'Hopper'
third, fourth = second, first
print(third, fourth)
```

3. Recall that a section of a string is called a slice. Work on the slices below.

```
element = 'oxygen'
print('first three characters:', element[0:3])
print('last three characters:', element[3:6])
```

- What is the value of element [:4]?
- What about element [4:]?
- Or element [:]?
- What is element [-1]?
- What is element [-2]?
- Given those answers, explain what element[1:-1] does.

4. Fix the capitalisation in messy\_string

```
messy_string = "OH mY, ThESE LEtters Are ALL OVER The PLace!"
```

Hint: Think about standardising the letters by e.g making all one case, then fixing the capitalization from there.

- 5. Check out the math package. https://docs.python.org/3/library/math.html and import it.
- What is the arc sine of -1, 0, 1 in radians?
- How many degrees in  $\arcsin(-1)$  radians?

# 2 Data Structures

#### 1. Questions:

- How do I arrange and process lots of items of data in Python?
- 2. Objectives:
- Create and work with lists and dicts
- 3. Keypoints:
- A list is a collection object that is a linear collection of other objects
- A dict is a collection object that is a group of key objects that point to values

Data structures are collection types that group lots of data into a single object and make working with lots of data easier. Python has some built in data structures we can use. In this section we'll look at using some of the most common and fundamental ones. As you gain exeperience with Python, you'll find that many other more complex data structures are made out of these.

## 2.1 Lists are ordered linear collections of items

The simplest data structure is a list. We can create a list simply by enclosing our data in square brackets.

my\_list = [1,3,5,7]
print( my\_list )

[1, 3, 5, 7]

More often, though, we'll get a list as the result of a function. Recall the random library we used earlier, it can sample k items from a list.

```
import random
selected = random.sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=3)
print( selected )
```

['red', 'red', 'red']

Here we get a list of integers. The point being it is now a collection of data that we can refer to as a single entity.

Lists can mix up any sort of data type,

```
numbers_and_letters = [1,2, 'three', 'IV', 5.0 ]
print( numbers_and_letters )
```

```
[1, 2, 'three', 'IV', 5.0]
```

including other lists.

```
list_of_lists = [ [1,2,3], ["a","b","c"] ]
print( list_of_lists )
```

```
[[1, 2, 3], ['a', 'b', 'c']]
```

## 2.1.1 List use

List elements can be accessed using indexing, like with strings.

```
print( numbers_and_letters[0] )
print( numbers_and_letters[2:4] )
```

```
1
['three', 'IV']
```

Indexing an element returns the whole element - so if that element happens to be a list itself - you get a whole list back

```
print( list_of_lists[1] )
```

['a', 'b', 'c']

To get at a single element in this case you must use multiple sets of square brackets, one for each list.

```
print( list_of_lists[0][1] )
print( list_of_lists[1][0] )
```

2 a

## 2.1.2 List methods

As they're still a regular Python object, just like strings or they have lots of methods. Much of what you'll want to do with a list is accomplished with methods. Again, there are many - here's a list of useful ones https://www.w3schools.com/python/python\_ref\_list.asp. As a single example from the list, let's look at .append(), which is used to stick something on the right hand side of the list.

```
numbers_and_letters.append("ninety")
print( numbers_and_letters )
```

[1, 2, 'three', 'IV', 5.0, 'ninety']

## 2.2 Dictionaries are unordered collections

Another very common data structure is a dictionary. A dictionary is a data structure that has many unique keys, each of which refers to a bit of other data called a value. We can construct them using the curly brackets and the key/value pairs

```
my_dict = {
    "key1" : "value1",
    "key2" : "value2"
}
```

```
print( my_dict )
{'key1': 'value1', 'key2': 'value2'}
```

Note the order in the dictionary isn't preserved. The information is stored according to an internal algorithm, not the order the information is added. We can use the square brackets to get a single value, but as a dictionary has no order and therefore no numeric index, we must use the key as the index.

print( my\_dict["key1"] )

value1

Dictionaries are useful when you want to ask for a bit of data by some name, rather than by its position in a list.

Dictionaries can hold anything in their values. But keys are restricted to particular datatypes. Strings and numbers are good keys, lists are not allowed.

```
print( { ["list_key", 1, 2] : ["some data"] } )
```

Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'

#### 2.2.1 Dictionary Methods

As with lists there are methods on Python Dictionaries. Here's a useful list https://www.w3schools.com/python/python\_ref\_dictionary.asp. These are mostly related to getting items on and off the list, we'll see some when we use dictionaries in loops later.

## 2.3 Quiz

1. Given the list below, use slicing to access only the last 2 entries.

- 2. Modify list\_for\_slicing using the .reverse() method. Verify that it is reversed as you expected.
- 3. Can you work out how to correct the wrong data in the dictionary below? Try to think of a way that *doesn't* involve re-writing the whole dictionary. Hint: can you assign straight to a key?

```
seasons = {
   'spring' : ['mar', 'apr', 'may'],
   'autumn' : ['jun', 'jul', 'aug'],
   'winter' : ['dec', 'jan', 'feb']
}
```

4. Add in the missing season.

# 3 Making Choices and Controlling Program Flow

- 1. Questions:
- How do we get Python to make choices?
- 2. Objectives:
- Use and understand if and else
- Use and understand boolean condition operators
- 3. Keypoints:
- Conditions let us test a value and run different bits of code dependently
- Whitespace is an important structuring element in Python code.

How can we use Python to automatically recognize differences in data such that it can change what code is run depending on the data and take a different action for each? In this chapter, we'll learn how to write code that runs only when certain conditions are true.

## 3.1 Conditionals check values and adjust which code is run

We can ask Python to take different actions, depending on a condition, with an if statement:

```
num = 37
if num > 100:
    print('greater')
else:
    print('not greater')
```

not greater

The second line of this code uses the keyword if to tell Python that we want to make a choice. If the test that follows the if keyword is true, the body of the if (i.e., the lines indented underneath it) are executed. If the test is false, the body of the else is executed instead. Only one or the other is ever executed:

The diagram below shows how this choice is being made.



Figure 3.1: Executing a Conditional

## 3.2 Indentation in Python is Pretty Important

The layout of code in Python is actually pretty important. The whitespace is structural and the amount of it tells us something about where we are in a program. Whitespace is particularly important in if. By convention the indent under each new if or else should be four spaces.

if x > y: do\_something()

An if within an if needs further indentation - it must also be four spaces further in, so a total of eight spaces.

```
if x > 100:
    print("bigger than 100")
    if x < 120:
        print("but smaller than 120")
    else:
        print("and bigger than 120")
else:
    print("x is too small")
```

This rule propagates, so a third level would need twelve spaces etc.

Note how the code above is structured now. Code at the same indentation is in the same group of code - at the same level. We can easily see which **if** and **else** go together as pairs. This

is the point of using whitespace like this - it gives us clean and visually consistent programs which the designers of Python value.

A common gotcha is that you always have to undo the indent at the end of the block. The first bit of the rest of the code must always be fully at the left of the page or Python will interpret the code incorrectly.

These indentation rules apply in other places in Python code and we'll come across them in due course.

## 3.3 More ifs

Conditional statements don't have to include an **else**. If there isn't one, Python simply does nothing if the test is false:

```
num = 53
print('before conditional...')
if num > 100:
    print(num,' is greater than 100')
print('...after conditional')
```

```
before conditional...
...after conditional
```

We can also chain several tests together using elif, which is short for "else if". The following Python code uses elif to print the sign of a number.

```
num = -3
if num > 0:
    print(num, 'is positive')
elif num == 0:
    print(num, 'is zero')
else:
    print(num, 'is negative')
```

-3 is negative

Note that the if and elif bits are mutually exclusive. Only one of them ever gets executed.

### 3.3.1 Testing equality

Note that to test for equality we use a double equals sign == rather than a single equals sign = which is already used to assign values.

## 3.4 Logical operators in tests

Python has all the standard logical operators that let us combine tests. Most commonly there is and or. An and operator is only true if both parts are true:

```
if (1 > 0) and (-1 > 0):
    print('both parts are true')
else:
    print('at least one part is false')
```

at least one part is false

while or is true if at least one part is true:

```
if (1 < 0) or (-1 < 0):
    print('at least one test is true')</pre>
```

at least one test is true

### 3.4.1 What is True and what is False

True and False are special words in Python called booleans, which represent truth values. A statement such as 1 < 0 returns the value False, while -1 < 0 returns the value True.

True and False booleans are not the only values in Python that are true and false. In fact, *any* value can be used in an if or elif.

```
if '':
    print('empty string is true')
if 'word':
    print('word is true')
if []:
    print('empty list is true')
if [1, 2, 3]:
```

```
print('non-empty list is true')
if 0:
    print('zero is true')
if 1:
    print('one is true')
word is true
non-empty list is true
one is true
```

It may seem strange to set things up this way, but in Python in practice it allows for some nice and easy to read and write constructions.

# 3.5 That's not not what I meant

Sometimes it is useful to check whether some condition is not true. The Boolean operator **not** can do this explicitly.

```
if not '':
    print('empty string is not true')
if not 'word':
    print('word is not true')
if not not True:
    print('not not True is true')
```

empty string is not true not not True is true

# 3.6 Quiz

1. Consider this code:

```
if 4 > 5:
    print('A')
elif 4 == 5:
    print('B')
elif 4 < 5:</pre>
```

```
print('C')
```

Which of the following would be printed if you were to run this code? Why did you pick this answer?

• A

- B
- C
- B and C

2. Consider this code:

```
if 4 > 5:
    print('A')
if 4 <= 5:
    print('B')
if 4 < 5:
    print('C')
```

Which of the following would be printed if you were to run this code? Why did you pick this answer?

• A • B

- C
- B and C

3. Consider this code:

```
if 4 > 5:
    print('A')
elif 4 <= 5:
    print('B')
elif 4 < 5:
    print('C')
```

Which of the following would be printed if you were to run this code? Why did you pick this answer?

- A
- B
- C
- B and C

# 4 Repeating actions with Loops

```
1. Questions:
```

- How can we repeat code an arbitrary number of times?
- 2. Objectives:
- Understand the for loop and the while loop
- Looping a specified number of times and with a range of numbers using range()
- 3. Keypoints
- for loops repeat code for each element in a collection
- while loops repeat code until a condition changes

To do that, we'll have to teach the computer how to repeat things.

An example task that we might want to repeat is printing each character in a word on a line of its own.

We can access a character in a string using its index. For example, we can get the first character of the word 'lead', by using word[0]. One way to print each character is to use four print statements:

```
word = 'lead'
print(word[0])
print(word[1])
print(word[2])
print(word[3])
```

l e a d

This is a bad approach for two reasons:

- 1. It doesn't scale: if we want to print the characters in a string thats hundreds of letters long, we'd be better off just typing them in.
- 2. It's fragile: if we give it a longer string, it only prints part of the data, and if we give it a shorter one, it produces an error because we're asking for characters that don't exist.

IndexError: string index out of range

## 4.1 For Loops

Instead we can use a loop - a construct that moves through a collection of data taking one bit at a time. Here's a loop in action

```
word = 'lead'
for char in word:
    print(char)
```

l e a d

This is shorter, certainly shorter than something that prints every character in a hundred-letter string and more robust as well. Note the indentation rules apply in the **for** loop.

Also, see how the same code works if we change the length of the word

```
word = 'oxygen'
for char in word:
    print(char)
o
x
y
g
e
n
```

The improved version uses a for loop to repeat code in this case print(), once for each thing in a sequence. The general form of a loop is:

Using the oxygen example above, the loop might look like this:



Figure 4.1: loop\_image

where each character (char) in the variable word is looped through and printed one character after another. The numbers in the diagram denote which loop cycle the character was printed in (1 being the first loop, and 6 being the final loop).

We can call the loop variable anything we like, but there must be a colon at the end of the line starting the loop, and we must indent anything we want to run inside the loop. Unlike many other languages, there is no command to signify the end of the loop body (e.g. end for); what is indented after the for statement belongs to the loop.

### 4.1.1 Choosing loop variable names

In the example above, the loop variable was given the name **char** as a mnemonic; it is short for 'character'. We can choose any name we want for variables. We might just as easily have chosen the name **banana** for the loop variable, as long as we use the same name when we invoke the variable inside

```
word = 'oxygen'
for banana in word:
    print(banana)
```

e n

о х у g

It is a good idea to choose variable names that are meaningful, otherwise it would be more difficult to understand what the loop is doing.

Here's another loop that repeatedly updates a variable:

```
length = 0
for vowel in 'aeiou':
    length = length + 1
print('There are', length, 'vowels')
```

There are 5 vowels

It's worth tracing the execution of this little program step by step.Since there are five characters in 'aeiou', the statement on line 3 will be executed five times. The first time around,length is zero (the value assigned to it on line 1) and vowel is 'a'.

The statement adds 1 to the old value of length, producing 1, and updates length to refer to that new value.

The next time around, vowel is 'e' and length is 1, so length is updated to be 2. After three more updates, length is 5; since there is nothing left in 'aeiou' for Python to process, the loop finishes and the print statement on line 4 tells us our final answer.

## 4.1.2 Loop variable lifespan

Note that a loop variable is just a variable that's being used to record progress in a loop. It still exists after the loop is over, and we can re-use variables that were previously defined as loop variables:

```
letter = 'z'
for letter in 'abc':
    print(letter)
print('after the loop, letter is', letter)
```

a b c after the loop, letter is c

This means your loop variable should ideally not be something you intend to use elsewhere.

Note also that finding the length of a string is such a common operation that Python actually has a built-in function to do it called len:

print(len('aeiou'))

### 5

#### 4.1.3 Looping over a list

Looping over a Python list is very common. The syntax is identical.

```
numbers_and_letters = [1,2, 'three', 'IV', 5.0 ]
for item in numbers_and_letters:
    print(item)

1
2
three
IV
5.0
```

#### 4.1.4 Looping over a range of numbers

What if we don't want to do every item in a collection, or if we want to do something a set number of times? We can create a collection that has the things we need.

Python has a built-in function called range() that creates a sequence of numbers. range() can accept 1, 2, or 3 parameters.

- If one parameter is given, range creates an array of that length, starting at zero and incrementing by 1. For example, range(3) produces the numbers 0, 1, 2.
- If two parameters are given, range starts at the first and ends just before the second, incrementing by one. For example, range(2, 5) produces 2, 3, 4.
- If range is given 3 parameters, it starts at the first one, ends just before the second one, and goes up in steps of the third one. For example range(3, 10, 2) produces 3, 5, 7, 9.

#### 4.1.5 Looping over a dictionary

You can loop over the keys in the dictionary in the same way as for the list, explicitly accessing the value using the key.

```
d = {'x': 1, 'y': 2, 'z': 3}
for key in d:
    print(key, 'has value', d[key])
x has value 1
y has value 2
```

```
z has value 3
```

Note that as dicts are intrinsically disordered, unlike lists the order they will be accessed in this way is arbitrary. If you care about order you can make a list of the keys using the keys() method and loop over that instead. Here we make a list of keys and sort it, then loop over the sorted key. Note the difference in the result to that above.

```
d = {'x': 1, 'y': 2, 'z': 3}
frozen_keys = sorted(d.keys())
for key in frozen_keys:
    print(key, 'has value', d[key])
x has value 1
y has value 2
```

z has value 3

Python can also be made to give you the key - value pairs if you want them, so you don't need to explicitly get the value each time. For this we use the .items() method and two loop variables. The first loop variable gets the key, the second the value.

```
d = {'x': 1, 'y': 2, 'z': 3}
for key, value in d.items():
    print(key, 'has value', value)
x has value 1
y has value 2
z has value 3
```

## 4.2 While Loops

A different sort of loop is the while loop. This loop repeats *while* something is in some state - usually the True state.

```
i = 1
while i < 6:
    print(i)
    i += 1</pre>
```

The while loop is somewhat rare in Python, but does get used from time to time.

# 4.3 Quiz

- 1. Using range, write a loop to print the first 3 positive integers.
- 2. Exponentiation is built into Python, 3<sup>2</sup> would be 3 **\*\*** 2. Write a loop that calculates the same result as 5 **\*\*** 3 using multiplication (and without exponentiation).

print(5 \*\* 3)

#### 125

- 3. Knowing that two strings can be concatenated using the + operator, write a loop that takes a string and produces a new string with the characters in reverse order, so 'Newton' becomes 'notweN'.
- 4. The built-in function enumerate() takes a sequence (e.g. a list) and generates a new sequence of the same length. Each element of the new sequence is a pair composed of the index (0, 1, 2,...) and the value from the original sequence:

```
fruits = ['apple', 'banana', 'grapes', 'pear']
for position, name in enumerate(fruits):
    print("The ", position, "fruit is ", name)
```

The 0 fruit is apple The 1 fruit is banana The 2 fruit is grapes The 3 fruit is pear

The function shuffle() in the random package rearranges a list in place (meaning it changes the original object, so you don't have to use a fresh variable name. Use the enumerate() and random.shuffle() to mix up the list below and work out where the digit 100 appears in the list.

```
big_numbers = list(range(1000) )
```

# **5** User Functions

#### 1. Questions:

- How can I re-use bits of my own code?
- 2. Objectives:
- Create and use our own functions
- Compose our own functions from other functions we made
- 3. Keypoints:
- Functions help us keep our programs concise and compact
- Functions make coding much easier by reducing debugging and allowing us to re-use code we've already written

In order to write concise readable and bug-free programs its a good idea to do as little coding as possible in a program. To this end we follow a principle called DRY - don't repeat yourself! Meaning that we don't re-write code that does the same stuff in mulitple places in a program, we put it in one place and refer to it from there. This is the basis of good programming

## 5.1 Building your own function

So we'd like a way to package our code so that it is easier to reuse and Python provides for this by letting us define our own functions. In use these work just like all the other functions we've already seen.

Let's look at a user function by defining a function fahr\_to\_celsius that converts temperatures from Fahrenheit to Celsius:

def fahr\_to\_celsius(temp):
 return ((temp - 32) \* (5/9))

The function definition opens with the keyword def followed by the name of the function (fahr\_to\_celsius) and a list of parameter names in brackets (temp). The parameter names are actually variables that carry the data given in the function call. The body of the function, the statements that are executed when it runs is indented below the definition line. The body



Figure 5.1: The Blueprint for a Python Function

concludes with a **return** keyword followed by the return value, the thing the function sends back to whatever called it.

When we call the function, the values we pass to it are assigned to those parameter variables so that we can use them inside the function. Inside the function, we use a return statement to send a result back to the code that asked for it.

Let's try running our function.

fahr\_to\_celsius(32)

#### 0.0

This command should call our function, using "32" as the input and return the function value.

In fact, calling our own function is no different from calling any other function:

```
print('freezing point of water:', fahr_to_celsius(32), 'C')
print('boiling point of water:', fahr_to_celsius(212), 'C')
```

freezing point of water: 0.0 C boiling point of water: 100.0 C

We've successfully called the function that we defined, and we have access to the value that we returned.

# 5.2 Composing functions

Now that we've seen how to turn Fahrenheit into Celsius, we can also write the function to turn Celsius into Kelvin:

```
def celsius_to_kelvin(temp_c):
    return temp_c + 273.15
print('freezing point of water in Kelvin:', celsius_to_kelvin(0.))
```

freezing point of water in Kelvin: 273.15

What about converting Fahrenheit to Kelvin? We could write out the formula, but we don't need to. Instead, we can compose the two functions we have already created:

```
def fahr_to_kelvin(temp_f):
    temp_c = fahr_to_celsius(temp_f)
    temp_k = celsius_to_kelvin(temp_c)
    return temp_k
print('boiling point of water in Kelvin:', fahr_to_kelvin(212.0))
```

boiling point of water in Kelvin: 373.15

This is our first taste of how larger programs are built, we define basic operations, then combine them in ever-large chunks to get the effect we want.

# 5.3 Variables inside and outside Functions

The function is insulated from the rest of the program. Things that happen in there don't affect what goes on elsewhere. Meaning we can re-use variable names inside the function that we used elsewhere without polluting them. Look what happens when the following piece of code is run

```
f = 0
k = 0
def f2k(f):
    k = ((f-32)*(5.0/9.0)) + 273.15
```

```
return k
print( f2k(8) )
print( f2k(41) )
print( f2k(32) )
print(k)
259.81666666666666
287.15
273.15
0
```

 ${\tt k}$  is 0 because the  ${\tt k}$  inside the function f2k doesn't know about the k defined outside the function.

## 5.4 Designing programs to use functions

It's a good programming strategy to think about building functions into programs. Try to look for places where you're doing the same set of commands over again - that would be a prime target for turning into a function. Think also about places where you could just refer to an operation by name - something like: count\_gc for example. If you spot places like this its a good idea to put the code into a function since it will make your program more readable. When you find you're re-using the same functions in mulitple programs, then you're ready to start putting them in external files and re-using them from there.

## 5.5 Quiz

1. "Adding" two strings produces their concatenation 'a' + 'b' is 'ab'. Write a function called **fence** that takes two parameters called **original** and **wrapper** and returns a new string that has the wrapper character at the beginning and end of the original. A call to your function should look like this:

print(fence('name', '\*'))

2. Note that **return** and **print** are not interchangeable.**print** is a Python function that *prints* data to the screen. It enables us, *users*, see the data. **return** statement, on the other hand, makes data visible to the program.

Let's have a look at the following function:

```
def add(a, b):
    print(a + b)
```

What will we see if we execute the following commands?

```
A = add(7, 3)
print(A)
```

3. If the variable s refers to a string, then s[0] is the string's first character and s[-1] is its last. Write a function called **outer** that returns a string made up of just the first and last characters of its input. A call to your function should look like this:

```
print(outer('helium'))
```

4. Rescaling an Array

Write a function **rescale** that takes a list as input and returns a corresponding list of values scaled to lie in the range 0.0 to 1.0.

(Hint: If L and H are the lowest and highest values in the original array, then the replacement for a value v should be (v-L) / (H-L). The max() function returns the biggest value in a list)

5. Consider this code:

```
a = 3
b = 7
def swap(a, b):
    temp = a
    a = b
    b = temp
swap(a, b)
print(a, b)
```

Which of the following would be printed if you were to run this code? Why did you pick this answer?

# 6 Working with Python Packages

#### 1. Questions:

• How can I use the wide range of code that other people have already written?

#### 2. Objectives:

- Install and Import packages using package managers
- Use vcfpy a genomics related package
- Examine objects from this package

#### 3. Keypoints:

- Packages that come from external sources need to be installed
- There are many, many packages
- Packages provide methods and objects of varying types
- Good package use relies on good package documentation

Packages are an important part of the Python 'ecosystem' that let us use a wide variety of code written by other people. Good packages provide functions, objects and methods that make working in a particular problem domain a lot easier.

# 6.1 Using and Installing Packages.

We've already seen how to load in packages using the import statement. Here we load the builtin sys package which has methods for working with the underlying system - the platform is the name of the operating system this code is running on.

```
import sys
print( sys.platform )
```

darwin

Some packages, like **sys** come as standard when Python is installed, some come from external sources and must be installed individually. There are multiple methods of installing packages. Assuming you set up Python using Anaconda for this book then you can use **conda** itself. Another more general way of installing packages is from PyPI, the Python Package Index.

None of these are done from within Python itself, they must be done from the regular commandline. The general form is:

```
conda install package_name
pip install package_name
```

# 6.2 The vcfpy Package

VCF files (variant call format) are files that describe SNPs and small indels in alignments from high-throughput sequencing data. The vcfpy package provides a lot of functionality for working with such files and the record they contain.

### 6.2.1 VCF files - a brief introduction

A VCF file is a column format file where each row represents a SNP/Indel record and the columns represent things describing it like Chromosome, Position, Reference Genome Allele, Alternative Genome Allele etc. Here's what one record looks like, split over a few lines.

Table 6.1: Table continues below

CHROM	POS	ID	REF	ALT	QUAL	FILTER
20	14370	rs6054257	G	А	29	PASS

Table 0.2. Table commutes below
---------------------------------

INFO	FORMAT	NA00001	NA00002
NS=3;DP=14;AF=0.5;DB;H2	GT:GQ:DP:HQ	0 0:48:1:51,51	1 0:48:8:51,51

NA00003	
1/1:43:5:.,.	

There's a lot of information in that one record, and each file has many tens of thousands of records! We wouldn't want to try and process tens of thousands manually. Let's look at loading in a file and looping through each record using the vcfpy package.

We've already seen that packages have functions we can call, and that doing so can sometimes return objects of new types. We'll use that pattern now to start processing a VCF file.

#### 6.2.2 The vcf.Reader object

The first package function we'll need to use is vcf.Reader which opens and connects to a file but doesn't do anything with it. It gives us a reader object we can use to access the file. We just need the file name that we wish to open.

```
import vcfpy
vcf_reader = vcfpy.Reader(open('/Users/macleand/Desktop/example.vcf', 'r'))
```

Now we can go ahead and extract the VCF records by using the reader object we created in a loop

```
for record in vcf_reader:
    print( record )
    print(type(record))
```

```
Record('20', 14370, ['rs6054257'], 'G', [Substitution(type_='SNV', value='A')], 29, ['PASS']
<class 'vcfpy.record.Record'>
Record('20', 17330, [], 'T', [Substitution(type_='SNV', value='A')], 3, ['q10'], {'NS': 3, 'I'
<class 'vcfpy.record.Record'>
Record('20', 1110696, ['rs6040355'], 'A', [Substitution(type_='SNV', value='G'), Substitution
<class 'vcfpy.record.Record'>
Record('20', 1230237, [], 'T', [], 47, ['PASS'], {'NS': 3, 'DP': 13, 'AA': 'T'}, ['GT', 'GQ'
<class 'vcfpy.record.Record'>
Record('20', 1234567, ['microsat1'], 'GTCT', [Substitution(type_='DEL', value='G'), Substitution('Substitution')
```

This does the code loop for every record in the file. And note how print() gives only a rough printout of the object, not every bit of information.

### 6.2.3 The vcf.Record object

Every time we loop we get a new vcf.Record object. This is a special sort of object provided by the vcfpy package that represents the VCF record. We already saw how objects have methods - special functions that apply straight to the data in that object. PyVCF is no exception. As well as methods, objects can have properties called attributes.

### 6.2.4 Object Attributes

As well as methods, objects have things called attributes.

Attributes of an object are different from methods of an object in that they tend to be things that just are rather than things that are computed. So a hypothetical **shape** object representing a geometric shape might have an attribute called **sides**. This wouldn't need recomputing as it would be the same every time.

Attributes are easy to spot as they use the object dot . syntax, omitting the brackets at the end of the attribute name. Let's examine that using the .POS attribute.

```
vcf_reader = vcfpy.Reader(open('/Users/macleand/Desktop/example.vcf', 'r'))
for record in vcf_reader:
    print( record.POS )
14370
```

Here we can see that we get the position of each SNP/Indel in the chromosome from .POS. Note that we had to redo the vcf\_reader creation - this is because the vcf.Reader object has in a sense "got to the end" of the data the first time we used it. It needs re-winding to the start and one way to do that is to recreate it and thereby reset it.

#### 6.2.4.1 Finding the methods and attributes an object has

If we want to find out what methods or attributes an object has we can use the dir() function which gives us these for an object.

```
print( dir(vcf_reader) )
```

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_enter\_\_', '\_\_eq\_\_', '\_\_exit

So the list is really large! Some methods and attributes begin with  $\_$  underscore characters. By convention these are things that are used internally by the object - so you don't need to worry about those. We can filter the ones we can use, by building a list of ones that don't start with '\_'

```
attrs = dir(vcf_reader)
keepers = []
for a in attrs:
    if not a.startswith("_"):
        keepers.append(a)
print( keepers )
```

['close', 'fetch', 'from\_path', 'from\_stream', 'header', 'parsed\_samples', 'parser', 'path',

This is a much shorter list. You can see the online documentation for the object here https://vcfpy.readthedocs.io/en/stable/api\_io.html#vcfpy-reader

Pretty much all objects and packages will have some online documentation so you can *usually* find something. Sometimes it's a bit sparse, unfortunately.

# 6.3 Quiz

- 1. Using PyVCF, calculate how many of the records are SNPs and how many are Indels. Use the documentation to see how you might achieve this https://vcfpy.readthedocs.io/en/stable/api\_record.h record
- 2. For each SNP, work out whether it's an A-G or C-T, a so-called transition. Would your method be efficient on a file of thousands of SNPS?
- 3. Which record has the greatest number of alleles? Would your method be efficient on a file of thousands of SNPs/Indels?
- 4. How many alternative alleles are heterozygous, for each record?

# 7 Putting code into scripts

#### 1. Questions:

- How do I write my code into a script that can be run again?
- 2. Objectives:
- Learn the conventions for creating script files.
- Get parameters from the command line and use them in scripts.
- 3. Keypoints:
- Scripts are files that contain programs of Python code.

Scripts are files full of code that has been put together in order to do a particular task. The idea being that the code will get re-run many times and not just as a one off.

Building a script is usually pretty easy - just type the code in to a text file.

You'll need a text editor, a program that deals with text but not in the same way as a word processor. Many such programs are available, try the options below

- 1. Atom (macOS/Windows) https://atom.io/
- 2. Notepad++ (Windows) https://notepad-plus-plus.org/

## 7.1 The script header

Most Python scripts have this on the first line

#### #!/usr/bin/env python

This is a Unix/Linux convention that affects how those systems interpret the file. Leave it in for convention's sake.

## 7.2 The filename extension

By convention, Python script filess end in the extension, .py.

So if we have the following in a file called hello\_world.py, we have a Python script.

#!/usr/bin/env python

print("Hello, World!")

## 7.3 Running a python script from the command line

Once created, the script itself is run from the python command on the command line. python is just a regular program that accepts a python script filename as its argument and runs the code in the script.

Here's an example terminal session that runs a script.

Last login: Thu Dec 6 10:59:32 on ttys000 ~/Desktop macleand\$ python hello\_world.py

Hello, World!

~/Desktop macleand\$

## 7.4 Getting Options from the Command Line

A good reason for creating scripts is because you want to be able to re-run the code in them. Sometimes you'll want to change some aspect or behaviour of the code according to settings given on the command-line. For example, you might want to work on a different input file each time. We can access the text from the command-line in the script, using the sys.argv attribute in the sys module.

Imagine the command line

python my\_script.py OPTION\_1 option\_2

We access it like this

```
#!/usr/bin/env python
import sys
print( sys.argv )
```

['/Users/macleand/Desktop/programming\_with\_python/renv/python/virtualenvs/renv-python-3.8/lil

```
['my_script.py', 'OPTION_1', 'option_2']
```

The sys.argv attribute gives us a list of the command line options. The first item in the list is the script name, the options come after that. So we can access the options by indexing the sys.argv list

python my\_script.py OPTION\_1 option\_2
#!/usr/bin/env python
import sys
first\_option = sys.argv[1]
second\_option = sys.argv[2]
print(second\_option, first\_option)

/private/var/folders/22/kjdvv\_k14cj1m6hq5hl527qw0006zc/T/tmprw33n0iy.json -f

option\_2 OPTION\_1

# 7.5 Quiz

- 1. Create a script that writes 'Hello, World!' to the screen. Run it.
- 2. Create a script that takes one argument and prints that to the screen as You said.. <argument> replacing <argument> with the value you give on the command line.
- 3. Create a script that takes two numbers from the command line and adds them together and prints the result.

# 8 Putting It All Together

# 8.1 A Challenge in Python

This file SRR020192.fastq.gz contains next generation sequencing reads in fastq format.

Write a Python script that uses the BioPython package to

- 1. Count the reads in the fastq file
- 2. Filter out low-quality reads from the fastq file
- 3. Calculate how many reads are retained
- 4. Use at least one user-defined function

# A Installing what you need

# A.1 Prerequisites

No specific Python knowledge is assumed for this book, though you do need to install some software. Most of it can be done using your computer's graphical interface.

- 1. Python 3 via Anaconda
- 2. A reasonably recent web-browser
- 3. The vcfpy and biopython python packages
- 4. The files example.vcf and SRR020192.fastq.gz
- 5. A text-editor

# A.2 Installing Python 3 with Anaconda

Follow this link and install Python 3.x for your operating system. https://www.anaconda. com/distribution/

## A.2.1 Note for macOS users

Accept all of the defaults during installation

Here is a video tutorial https://www.youtube.com/watch?v=TcSAln46u9U

## A.2.2 Note for Windows users

Install Python 3 using all of the defaults for installation except make sure to check Add Anaconda to my PATH environment variable.

Here is a video tutorial https://www.youtube.com/watch?v=xxQ0mzZ8UvA

## A.2.3 Note for Linux Users

You'll need to be able to use the command-line to install with Anaconda. If you aren't comfortable with this, ask for assistance from the local support team.

- 1. Open https://www.anaconda.com/download/#linux with your web browser.
- 2. Download the Python 3 installer for Linux.
- 3. Open a terminal window. 4. Type bash Anaconda3-and then press Tab. The name of the file you just downloaded should appear. If it does not, navigate to the folder where you downloaded the file, for example with: cd Downloads. Then, try again.
- 4. Press enter. You will follow the text-only prompts. To move through the text, press the spacebar.
- 5. Type yes and press enter to approve the license.
- 6. Press enter to approve the default location for the files.
- 7. Type yes and press enter to prepend Anaconda to your PATH (this makes the Anaconda distribution the default Python).
- 8. Close the terminal window.

# A.3 Starting a Jupyter Notebook

## A.3.1 macOS

- 1. Start the Terminal application in Applications -> Utilities
- 2. Type jupyter notebook, it should start in your web browser

## A.3.2 Windows

1. From the Start menu, search for and open Anaconda 3 or Jupyter Notebook. You should be able to start a notebook directly by clicking the Jupyter Notebook icon.

## A.3.3 Linux

- 1. Open the terminal application. It is usually in the task bar or dock
- 2. Type jupyter notebook, it should start in your web browser

# A.4 Installing Python Packages with conda

You can use conda to install new Python packages using the Terminal by typing conda install conda conda install conda.

You can install the required packages with the following commands:

conda install vcfpy conda install biopython

Accept all defaults when the system asks a question.

# A.5 Installing a Text Editor

A text editor is a program that deals with text in a way that is appropriate to writing programs. It is quite different to a word processor. Many such programs are available, try the options below:

- 1. TextMate (macOS) https://macromates.com/
- 2. Atom (macOS/Windows) https://atom.io/
- 3. Notepad++ (Windows) https://notepad-plus-plus.org/

# **B** Acknowledgements

Some of the quizzes and examples in this book particularly those in Chapters 3 to 7 are taken from the Software Carpentry lesson on Programming in Python http://swcarpentry.github.io/python-novice-inflammation/. Other examples in Chapter 9 and 10 are taken from the Biopython tutorial http://biopython.org/DIST/docs/tutorial/Tutorial.html and the vcfpy tutorial https://vcfpy.readthedocs.io/.These are re-used under their respective licences.

The rest of the materials are licensed under Creative Commons 0.