Beginning Programming with Python

Dan MacLean

2019-01-08

Contents

1 Prerequisites **5**
- 1.1 Installing Python 3 with Anaconda . . . 5
- 1.2 Starting a Jupyter Notebook . . . 6
- 1.3 Installing Python Packages with `conda` . . . 6
- 1.4 Installing a Text Editor . . . 7

2 Motivation **9**

3 Working with Data **11**
- 3.1 Using variables as names for data . . . 12
- 3.2 Python has three main types of function. . . . 12
- 3.3 Objects and types in Python . . . 14
- 3.4 Quiz . . . 16

4 Data Structures **17**
- 4.1 Lists . . . 17
- 4.2 Dictionaries . . . 19
- 4.3 Quiz . . . 19

5 Making Choices and Controlling Program Flow **21**
- 5.1 Conditionals . . . 21
- 5.2 Indentation in Python . . . 22
- 5.3 More `ifs` . . . 22
- 5.4 Logical operators . . . 23
- 5.5 That's Not Not What I Meant . . . 24
- 5.6 Quiz . . . 24

6 Repeating actions with Loops **27**
- 6.1 For Loops . . . 28
- 6.2 While Loops . . . 31
- 6.3 Quiz . . . 32

7 User Functions **33**
- 7.1 Composing Functions . . . 34
- 7.2 Variables Inside and Outside Functions . . . 34
- 7.3 Designing Programs to use functions . . . 35
- 7.4 Quiz . . . 35

8 Working with Python Packages **37**
- 8.1 Using and Installing Packages. . . . 37
- 8.2 The PyVCF Package . . . 38
- 8.3 Quiz . . . 40

9 Putting code into scripts **41**
- 9.1 The Script Header . . . 41

9.2 The Filename Extension 42
9.3 Running a Python script 42
9.4 Getting Options from the Command Line 42
9.5 Quiz 43

10 Putting It All Together **45**
10.1 A Challenge in Python 45

11 Acknowledgements **47**

1 Prerequisites

No specific knowledge is assumed for this book, though you do need to install some software.

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

1.1 Installing Python 3 with Anaconda

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

1.1.1 Note for macOS users

Accept all of the defaults during installation

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

1.1.2 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

[1] assets/example.vcf
[2] assets/SRR020192.fastq.gz

1.1.3 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[3] 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.

1.2 Starting a Jupyter Notebook

1.2.1 macOS

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

1.2.2 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.

1.2.3 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

1.3 Installing Python Packages with `conda`

You can use `conda` to install new Python packages using the Terminal by typing `conda install <package_name>`.

You can install the required packages with the following commands:

```
conda install pyvcf
conda install biopython
```

Accept all defaults when the system asks a question.

[3]

1.4 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/[4]
2. Atom (macOS/Windows) https://atom.io/[5]
3. Notepad++ (Windows) https://notepad-plus-plus.org/[6]

4
5
6

2

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? Well, 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, it 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 that, but at least you'll know Python.

Happy Programming!

3

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.

huh?

Data is a very big word in programming, it carries a lot of meaning. It can mean very small bits of stuff, like a single text character, or huge amounts of stuff, like the images collected by the Hubble telescope. It's all the same to the computer and programming language.

3.1 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)
```

```
## Hello, world!
```

```
x = 100
print(x)
```

```
## 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)
```

```
## 65
```

```
print(weight_pounds)
```

```
## 143.0
```

3.2 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

3.2.1 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/fu
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.

[1]

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)
```

```
## 2
```

```
print(2 * 2)
```

```
## 4
```

```
print(3 - 3)
```

```
## 0
```

```
print(4 / 4)
```

```
## 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.

3.2.2 Package Methods

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 `numpy` numerical Python package that provides functions for doing fast, large scale numerical analysis.

```
import numpy
```

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 `arange()` which gives us a list of numbers.

```
numbers = numpy.arange(10)
print( numbers )
```

```
## [0 1 2 3 4 5 6 7 8 9]
```

3.2.3 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[2], which shows us all the String methods. This works well for finding all methods for objects of other types.

Round Up

We have discussed three types of function. The basic type is Python's built in functions, these are available anywhere in a Python program. The second, Package functions must be imported with package and accessed using the package name and dot. The third is the method which is a function tied to an object and is accessed using the variable name and the dot.

3.3 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

3.3.1 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
```

[2]

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.

3.3.1.1 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#tex

3.3.2 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 )
```

```
## 4.0
```

```
print( int( 10.0 / 3.0 ) )
```

```
## 3
```

[3]

3.4 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 string is called a slice.

```
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[4] and `import` it.

- What is the arc sine of `-1`, `0`, `1` in radians?
- How many degrees in $\arcsin(-1)$ radians?

[4]

4

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.

4.1 Lists

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 `numpy` function we used earlier.

```
import numpy
numbers = numpy.arange(15)
print( numbers )
```

```
## [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

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']]
```

4.1.1 List use

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

```
print( numbers_and_letters[0] )
```

```
## 1
```

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

```
## ['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] )
```

```
## 2
```

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

```
## a
```

4.1.2 List methods

Much of what you'll want to do with a list is accomplished with methods. Sticking something on the end is done with `.append()`

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

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

huh?

In the example above we didn't need to save this back to a variable. Note that this isn't a mistake. This method modified the object `in-place`. Some methods will do this, and at this stage which is which is going to seem somewhat arbitrary. The documentation will explain which do and don't.

4.2 Dictionaries

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 informaion 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'
```

4.3 Quiz

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

```
list_for_slicing = [["fluorine", "F"],
                    ["chlorine", "Cl"],
                    ["bromine", "Br"],
                    ["iodine", "I"],
                    ["astatine", "At"]]
```

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.

5

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.

5.1 Conditionals

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` statement 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.

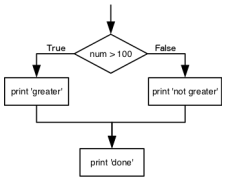

Figure 5.1: Executing a Conditional

5.2 Indentation in Python

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.

huh?

You are probably thinking that this is a bit of pain. Having to type in so many spaces is a bit tedious. For this reason, most text editors will allow you to use the `tab` key to enter four spaces instead of a genuine `tab`. They'll also show the spaces indented as little dots. Try playing about with the preferences in your text editor if you can't see this.

5.3 More `if`s

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...')
```

```
## before conditional...
```

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

5.3.1 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.

5.4 Logical operators

Python has all the standard logical operators that let us combine tests. Most commonly there is `and` 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')
```

```
## at least one test is true
```

5.4.1 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')
```

```
## word is true
```

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

```
## non-empty list is true
```

```
if 0:
     print('zero is true')
if 1:
    print('one 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.

5.5 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')
```

```
## empty string is not true
```

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

```
## not not True is true
```

5.6 Quiz

1. 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

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

6

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])
```

```
## l
```

```
print(word[1])
```

```
## e
```

```
print(word[2])
```

```
## a
```

```
print(word[3])
```

```
## 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.

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

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-3-7974b6cdaf14> in <module>()
      3 print(word[1])
      4 print(word[2])
----> 5 print(word[3])

IndexError: string index out of range
```

6.1 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:

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).

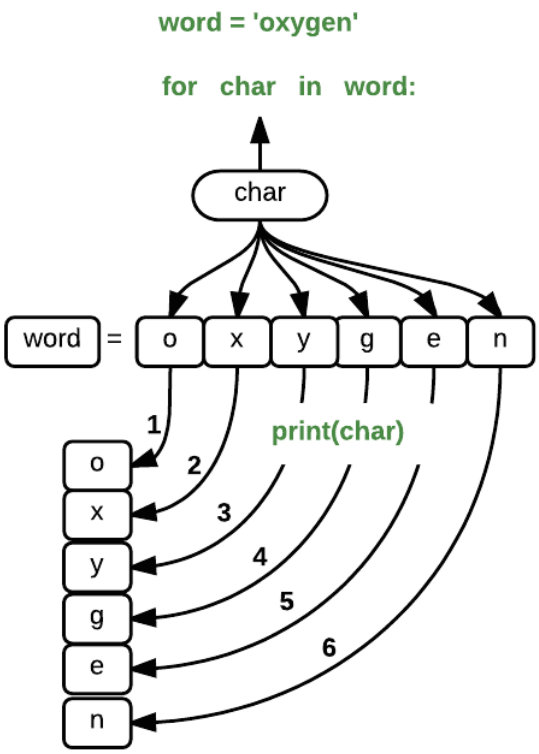

Figure 6.1: loop_image

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.

6.1.1 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)
```

```
## o
## x
## y
## g
## e
## n
```

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.

6.1.2 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)
```

```
## a
## b
## c
```

```
print('after the loop, letter is', letter)
```

```
## 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
```

6.1.3 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
```

6.1.4 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`.

6.1.5 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
```

6.2 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
```

```
## 1
## 2
## 3
## 4
## 5
```

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

6.3 Quiz

1. Using `range`, write a loop to print the first 3 positive integers.
2. Exponentiation is built into Python, 3^2 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) )
```

7

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. 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 start 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.

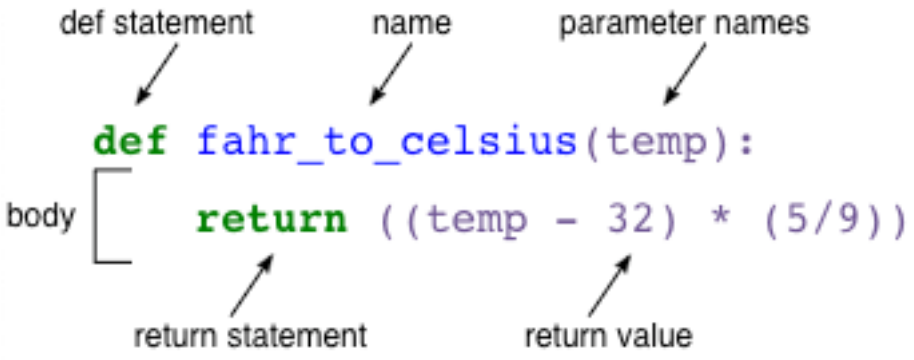

Figure 7.1: The Blueprint for a Python Function

The body 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)
```

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')
```

```
## freezing point of water: 0.0 C
```

```
print('boiling point of water:', fahr_to_celsius(212), '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.

7.1 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.

7.2 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
```

k is 0 because the `k` inside the function `f2k` doesn't know about the `k` defined outside the function.

7.3 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.

7.4 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?

1. 7 3
2. 3 7
3. 3 3
4. 7 7

8

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 PyVCF - 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.

8.1 Using and Installing Packages.

We've already seen how to load in packages using the `import` statement. Here we load the builit-in `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 command-line. The general form is:

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

8.2 The PyVCF Package

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

8.2.1 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 8.1: Table continues below

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

Table 8.2: Table continues 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 PyVCF 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.

8.2.2 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 vcf
vcf_reader = vcf.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(CHROM=20, POS=14370, REF=G, ALT=[A])
## <class 'vcf.model._Record'>
## Record(CHROM=20, POS=17330, REF=T, ALT=[A])
## <class 'vcf.model._Record'>
## Record(CHROM=20, POS=1110696, REF=A, ALT=[G, T])
## <class 'vcf.model._Record'>
## Record(CHROM=20, POS=1230237, REF=T, ALT=[None])
## <class 'vcf.model._Record'>
## Record(CHROM=20, POS=1234567, REF=GTCT, ALT=[G, GTACT])
## <class 'vcf.model._Record'>
```

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.

8.2.3 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 PyVCF 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.

8.2.4 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 everytime.

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

```
## 14370
## 17330
## 1110696
## 1230237
## 1234567
```

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.

8.2.4.1 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__', '__eq__', '__format__', '__ge__', '__
```

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 )
```

```
## ['alts', 'contigs', 'encoding', 'fetch', 'filename', 'filters', 'formats', 'infos', 'metadata', 'rea
```

This is a much shorter list. You can see the online documentation for the object here https://pyvcf.readthedocs.io/en/latest/API.html#vcf-reader[1]

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

8.3 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://pyvcf.readthedocs.io/en/latest/API.html#vcf-model-record[2]
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?

1

2

9

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

9.1 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.

1
2
3

9.2 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!")
```

9.3 Running a Python script

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$
```

9.4 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 )
```

```
['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)
```

```
option_2 OPTION_1
```

9.5 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.

10

Putting It All Together

10.1 A Challenge in Python

This file ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz[1] contains next generation sequencing reads in `fastq` format.

Write a Python script that use 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

[1]

11

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/[1]. Other examples in Chapter 9 and 10 are taken from the Biopython tutorial http://biopython.org/DIST/docs/tutorial/Tutorial.html[2] and the PyVCF tutorial https://pyvcf.readthedocs.io/[3].These are re-used under their respective licences.

1

2.

3