Introduction to Python

Simon Andrews, Steven Wingett
simon.andrews@babraham.ac.uk
@simon_andrews
v2021-10

Babraham Bioinformatics

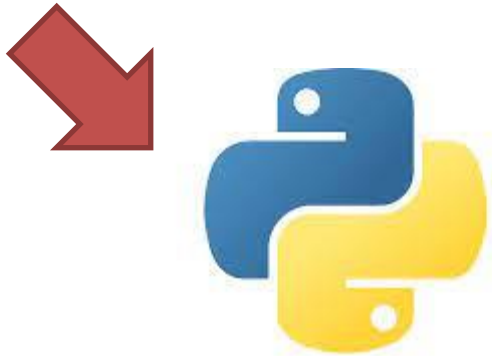
Setting up a Python Environment

Babraham Bioinformatics

Python is a ‘scripting’ language

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

print("I am a python program")
```

VS Code



python™

```
python
python.exe
python3
python3.exe
```

https://www.python.org/

```
C:\Introduction to Python>python example.py
I am a python program
```

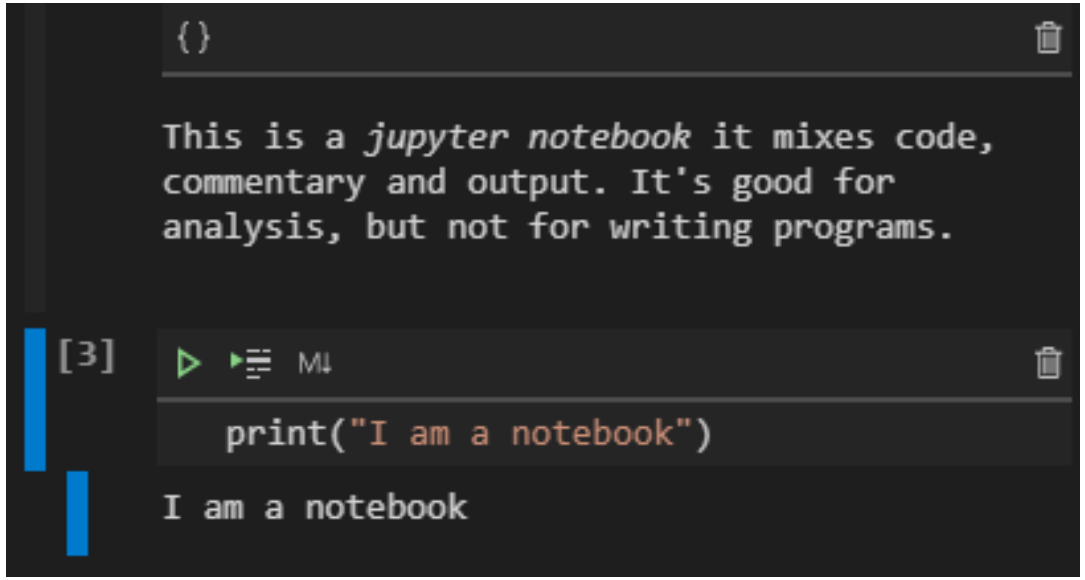
Different environments for writing python

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

print("I am a python program")
```

Scripted: code in text file, output in console

```
C:\Users\andrewss\>python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020,
17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or
"license" for more information.
>>>
>>> print("I am an interactive session")
I am an interactive session
>>>
```

Interactive: code and output in console

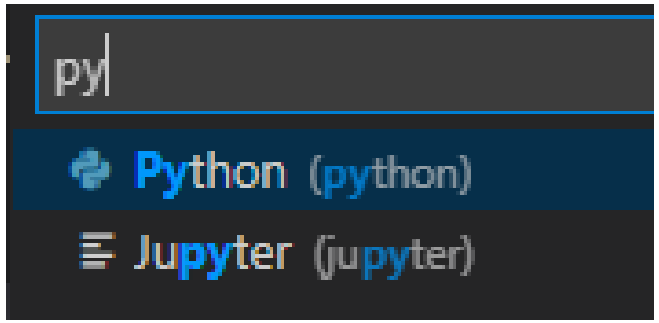


Notebook: code, commentary and output in a single file

Using VSCode to write a python script

- Install VSCode
- Install Python interpreter
- Open VSCode
  - File > New File

  - Select Python



  - File > SaveAs
    - Use a `.py` extension

Your first python program

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

my_name = "Simon"

print (my_name,"wrote his first python program")

print("He is very proud")
```

```
C:\>"C:/Program Files/Python39/python.exe" "c:/Introduction to Python/first_program.py"
Simon wrote his first python program
He is very proud
```

Python script basics

Where to find an interpreter

Comments use #

Series of python ‘statements’. One per line (generally). These are executed in order, from the top of the file to the bottom.

Your program finishes at the end of the file

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

# Create a variable with my name in it

my_name = "Simon"

print (my_name,"wrote his first python program")

print ("He is very proud")
```

Variables and Data Types

- A ‘variable’ is some data which you have given a name
- There are several different types of data structure
  - We’re starting with the ‘scalar’, a data type which holds a single value

- Python is a ‘dynamic’ but ‘strongly typed’ language
  - **Dynamic** = You don’t need to say what type of data a variable will hold when you create it (and you can change it at any time)

  - **Strongly typed** = Python tracks what type of data you have and changes its behaviour based on the type of the data

Creating a variable

- Variables are created or updated using the = operator
  - ‘Operator’ just means special symbol
  - Variable ‘types’ are determined by the data used
  - Python style guide says *"Variable names should be lowercase, with words separated by underscores as necessary to improve readability"*

```
x = 5         # x is an int (Integer, whole number)
x = 5.5       # x is a float (Floating point number, fractional)
x = True      # x is a bool (Boolean, logical True/False)
x = "Simon" # x is an str (String, piece of text)
```

```
x = input("What is your name? ") # Ask user for a str
```

Different ways to access functionality

- Operators
  - Special symbols to denote an operation (eg + * / etc)

```
5 + 10
```

- Functions
  - Named pieces of functionality into which data is passed

```
len("simon")
```

- Methods
  - Functions which are accessed via the data directly

```
"simon".upper()
```

Functions vs Methods

- Functions
  - Named pieces of code. All data (arguments) must be passed in to them. Accessed either in the core language or from packages

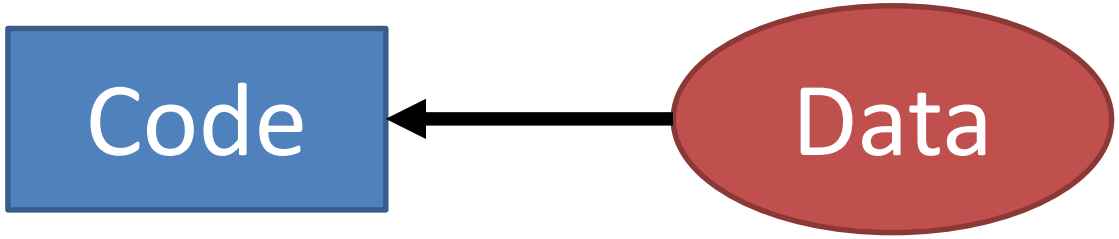
```
>>> len("Simon")
5
```

- Methods
  - Functions which are associated with a type of data (string, date etc). Called via the data, you don’t need to pass the data in to the method

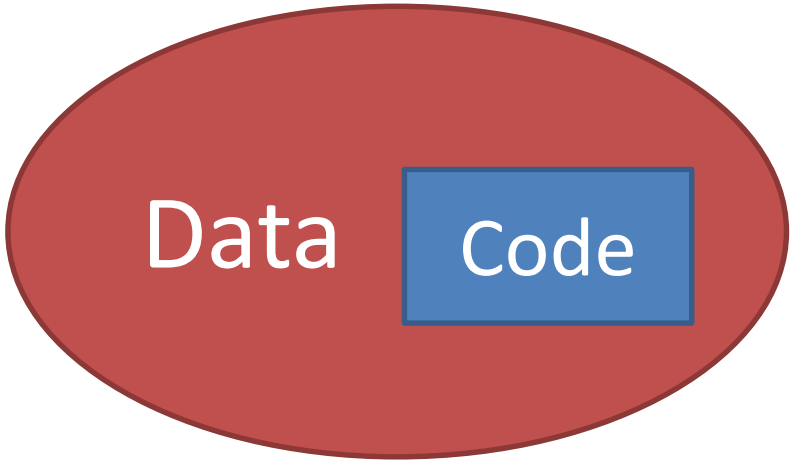
```
>>> "Simon".upper()
'SIMON'
```

Functions vs Methods

- Function



- Method



Functionality is linked to data type

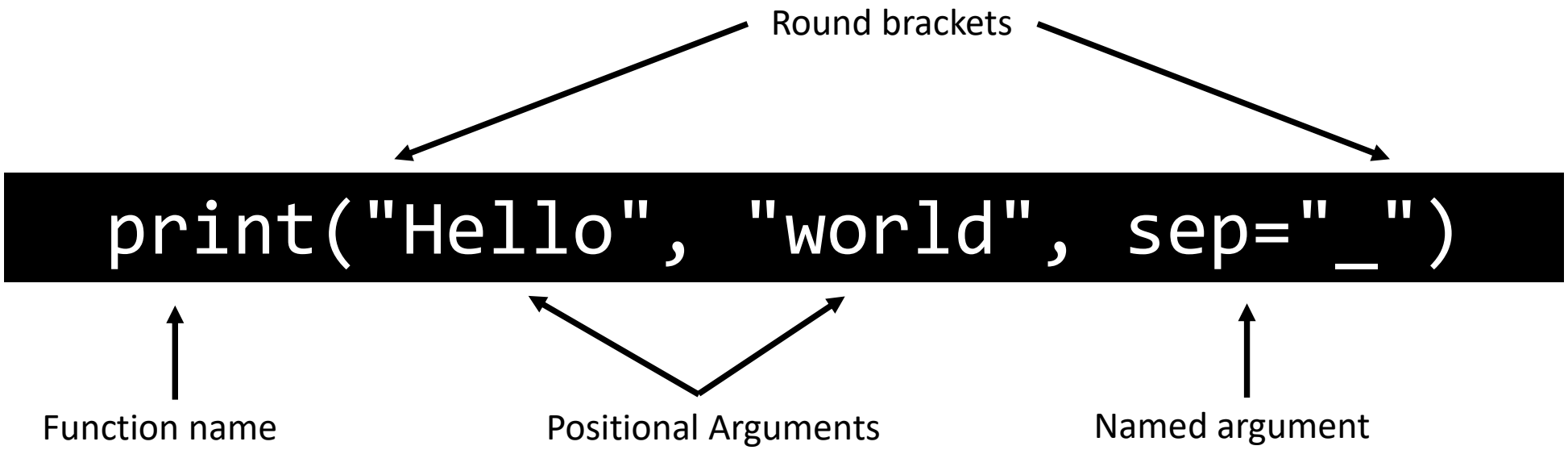
```
•  5  +  10        # 15
• "5" + "10"       # 510

• "5".upper()      # 5
•  5.upper()       # SyntaxError: invalid syntax

• str(5) + str(10)          # "510"
• float("5") + int(10)      # 15
```

Common Numeric Operators

Operator	Action
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Raise to a power
//	Floor division
%	Modulo

```
5 + 10
23 - 56
10 * 4.5    # Can mix int/float
20 / 7      # Converts to float
2 ** 5
20 // 7     # Stays as int
20 % 7      # Calculates remainder
```

Running a function



How to use a function?

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

List of built in functions

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Finding methods from data

```
>>> dir("some text")
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

>>> help("some text".lower)
Help on built-in function lower:

lower() method of builtins.str instance
    Return a copy of the string converted to lowercase.
```

Values looking like __name__ are special, automatically created variables or methods. They're mostly for internal use but we do sometimes use them directly

Finding methods via data type (class)

```
>>> type("simon")
<class 'str'>

>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
```

The python standard library

- Most functionality (functions / methods) is not in the core python language, but comes from extensions called 'packages'
- Python comes with an enormous collection of packages called the 'standard library' which are guaranteed to be present with any python installation
- Additional packages can be installed from the Python Package Index (pypi)

Text manipulation

`string` — Common string operations
`re` — Regular expression operations

Data Types

`datetime` — Basic date and time types
`zoneinfo` — IANA time zone support
`calendar` — General calendar-related functions
`array` — Efficient arrays of numeric values
`copy` — Shallow and deep copy operations
`pprint` — Data pretty printer
`graphlib` — Operate with graph-like structures

Numeric and Mathematical Modules

`math` — Mathematical functions
`random` — Generate pseudo-random numbers
`statistics` — Mathematical statistics functions

File and Directory Access

`os.path` — Common pathname manipulations
`stat` — Interpreting stat() results
`tempfile` — Generate temporary files and directories
`glob` — Unix style pathname pattern expansion
`shutil` — High-level file operations

Data Persistence

`pickle` — Python object serialization
`sqlite3` — DB-API 2.0 interface for SQLite databases

Data Compression and Archiving

`gzip` — Support for gzip files
`bz2` — Support for bzip2 compression
`zipfile` — Work with ZIP archives
`csv` — CSV File Reading and Writing

Generic Operating System Services

`os` — Miscellaneous operating system interfaces
`io` — Core tools for working with streams
`time` — Time access and conversions
`argparse` — Parser for command-line options

Internet Data Handling

`email` — An email and MIME handling package
`json` — JSON encoder and decoder

Graphical User Interfaces with Tk

`tkinter` — Python interface to Tcl/Tk

Software Packaging and Distribution

`distutils` — Building and installing Python modules
`venv` — Creation of virtual environments

Using functions from the standard library

Use functions via the package

```
import math
math.sqrt(10)

3.162277
```

Import individual functions

```
from math import sqrt
sqrt(10)

3.162277
```

```
import math as m
m.sqrt(10)

3.162277
```

```
from math import *
sqrt(10)

3.162277
```

Finding functions in a package

```
import math
help(math)

Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.
        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.
        The result is between -pi/2 and pi/2.
```

Also at: https://docs.python.org/3/library/math.html

Finding methods in a package

- Some packages define a new data type (class) and use that to call methods, rather than providing functions

```
CLASSES
    class Random(_random.Random)
     |  Random(x=None)
     |
     |  Random number generator base class used by bound module functions.
     |
     |  Used to instantiate instances of Random to get generators that don't
     |  share state.
...
     |  choice(self, seq)
     |      Choose a random element from a non-empty sequence.
```

Also at: https://docs.python.org/3/library/random.html

The random package has both methods and functions

```
CLASSES
    class Random(_random.Random)
     |  Random(x=None)
     |
     |  choice(self, seq)
     |      Choose a random element from a non-empty sequence.
```

```
FUNCTIONS
    choice(seq) method of Random instance
        Choose a random element from a non-empty sequence.
```

Example of using random

```
import random

# Use a function
print(random.randint(0,10))

# Use a method
# Make an instance of the Random datatype (class)
generator = random.Random()
# Call a method on this variable
print(generator.randint(0,10))
```

Example Script

- Input a name
- Input an age in years

- Output the year in which they were born
- Output the number of days they've been alive

- We're going to take some mathematical liberties ☺

```
#!C:\Program Files\Python39\python.exe

# A program to calculate someone's age
name = input("What is your name? ")
age = input("What is your age (in years)? ")

# Age starts as a string, so we need to convert it to be a number
age = int(age)

age_days = age * 365

# Import the time module so we can get the current year.
import time

year = time.gmtime().tm_year
born_in = year - age


print(name, "was borm in", born_in, "he is",age_days,"days old")
```

Exercise 1

Babraham Bioinformatics

Python Data Structures

Babraham Bioinformatics

Python Data Structures

- Holds a single value
  - `scalar`
- Holds multiple ordered values
  - `list, tuple`
- Holds multiple, unique, unordered values
  - `set`
- Lookup table, keys and values
  - `dictionary`

Lists

- Modifiable structure to hold an ordered set of data
- Values can be anything (scalars or other data structures)
- Lists can be created empty or with data in them
- You can add or remove data from a list, or extract subsets

```
empty_list = []

filled_list = [1,True,"simon",[5,6]]
```

List Methods

- `append` - Add something to the end
- `clear` - Remove all content
- `count` - Count the instances of a specific value
- `extend` - join lists together
- `index` - find the position of a value
- `insert` - Add data anywhere in the list
- `pop` - Remove the last value
- `remove` - Remove a specific value
- `reverse` - Reverse the list
- `sort` - Order the list

Note that these methods modify the list in place, they don't return anything.

List examples

```
my_list = ["dog","cat","gerbil"]                # dog cat gerbil

my_list.append("mouse")                         # dog cat gerbil mouse

my_list.extend(["cat","dog"])                   # dog cat gerbil mouse cat dog

my_list.count("cat")                            # 2

my_list.remove("gerbil")                        # dog cat mouse cat dog

my_list.insert(2,"rat")                         # dog cat rat mouse cat dog

last_value = my_list.pop()                      # dog cat rat mouse cat

my_list.index("dog")                            # 0

my_list.reverse()                               # cat mouse rat cat dog

my_list.index("dog")                            # 4
```

Accessing List Subsets

`my_list[start:end:step]`

- All positions start counting at 0 (everywhere in python)
  - Negative positions count back from the end of the list
  - Start is inclusive, end is exclusive
- You don’t have to supply all of the options
  - Later ones can just be omitted
  - Earlier ones can be omitted, but you still need the colons

Accessing List Subsets

my_list[start:end:step]

```
my_list = ["a","b","c","d","e","f","g","h"]

my_list[2]          # c
my_list[-2]         # g

my_list[0:5]        # a b c d e
my_list[:5]         # Same thing

my_list[0:5:2]      # a c e

my_list[3:]         # d e f g h

my_list[::3]        # a d g
my_list[::-1]       # h g f e d c b a
```

Changing Lists via Subsets

```
my_list[start:end:step] = x
```

```
my_list = ["a","b","c","d"]

my_list[0] = "X"                          # X b c d

my_list[1:3] = ["A","D","D","E","D"]  # X A D D E D d

my_list[2:2] = ["+","+"]                  # X A + + D D E D d
```

- Replace a single scalar, or a range
- Range replacements don't have to be the same size
- Zero-sized selections allow for insertions

Copying vs In-Place changes

- Python sometimes has two ways to do the same thing
  - Via a function or selection, returning a modified copy of the data
  - Via a method or replacement, changing the original copy of the data

- All python data (other than scalars) are 'references', which means that copying can be unintuitive

Copying vs In-Place changes

```
sorted(my_list)     # Returns a sorted copy of my_list

my_list.sort()      # Returns nothing, but sorts my_list in place



reversed(my_list)   # Returns a reversed copy of my_list*
my_list[::-1]       # Returns a reversed copy of my_list

my_list.reverse()   # Returns nothing. Reverses my_list in place
```

* Technically it returns an iterator over a reversed version of `my_list`, but we haven't talked about those yet

References, Shallow and Deep Copying

- All Python data structures (apart from scalars) are references
  - The data sits at some location in your computers memory
  - The variable holds the address of that location

- When you copy a variable you're not copying the data, you're copying the location of the data

- Making an actual data-level copy takes more work

References, Shallow and Deep Copying

```
my_list = ["dog","cat"]

my_copy = my_list

my_copy.append("mouse")

my_copy[-1]          # mouse
my_list[-1]          # ALSO mouse !!!
```

Shallow Copying

```
shallow_list = ["dog","cat","mouse"]

shallow_copy = shallow_list.copy()

shallow_copy[-1] = "gerbil"

# shallow_list is dog cat mouse
# shallow_copy is dog cat gerbil
```

Multi-Level Lists

- Lists can hold anything, even other lists
- This is a simple way to create multi-level data structures

```
multi_list = []

multi_list.append([10,20,30])                # Appending a list to a list
multi_list.append(["ten","twenty","thirty"]) # Note *not* the same as extend()
multi_list.append(["X","XX","XXX"])

multi_list           # [[10, 20, 30], ['ten', 'twenty', 'thirty'], ['X', 'XX', 'XXX']]
multi_list[1]        # ['ten', 'twenty', 'thirty']
multi_list[1][2]     # 'thirty'
```

Copying Multi-Level Lists

```
reference_copy = multi_list
top_level_copy = multi_list.copy()

from copy import deepcopy
deep_copy = deepcopy(multi_list)
```

Tuples

- Very similar to lists
  - Hold ordered sets of data
- Big difference is that tuples are 'immutable'
  - They can't be changed from the data they originally contain
- Most of the operations are the same as for lists
  - Selections return tuples rather than lists

Tuple examples

```
simple_tuple = (1,5,"fred") # Round brackets, not square
simple_tuple[:2]            # (1,5)

single_tuple = ("one",)     # Need trailing comma
```

Dictionaries

- The other big remaining data structure in core python
- Structured as a lookup table
  - **Key**: An index value (eg text or number) to look up by
  - **Value**: A data structure to link to that key

- Keys can be any immutable data type (strings, numbers or tuples)
- Keys must be unique
  - Values can be repeated under different keys

- Dictionaries are also ordered (they remember the order keys were added)*

* As of python v3.7 - before that they weren't

Dictionaries

```
# Creating dictionaries
empty_dict = {}                 # Curly brackets
populated_dict = {              # : between key and value
    "key1" : 10,                # , between value and next key
    "key2" : 20                 # Can also write on one line
}

# Retrieving values
populated_dict["key2"]      # Note SQUARE brackets to use

# Adding / replacing / removing values
empty_dict["simon"] = "Cambridge"      # Creates new key
populated_dict["key1"] = 1000          # Replaces old value
populated_dict.pop("key2")             # Removes (and returns)
```

Sets

- Sets are like dictionaries, but without values
- They hold a unique set of immutable keys
- They are very quick to look up what is (and isn't) in the set

- Sets are NOT ordered
  - There is an ordered-set package which provides this
  - It's not in the core package collection

Sets

```
# Creating a set
empty_set = set()                          # Must use a function, not {}
populated_set = {"oak","fir","ash"}        # No colons, just commas

# Adding / Removing
empty_set.add("simon")                 # Fine if it's already there
populated_set.remove("fir")            # Get an error if it's not there

# Testing
"ash" in populated_set          # True
"larch" in populated_set        # False
```

```python
#!python
existing_data = {
    "WT": [2,5,4,6],
    "KO": [8,6,9,12]
}

condition = input("Which condition? ")
value = float(input("What value? "))

existing_data[condition].append(value)

wt_count = len(existing_data["WT"])
ko_count = len(existing_data["KO"])

print ("There are",wt_count,"WT values:",existing_data["WT"])
print ("There are",ko_count,"KO values:",existing_data["KO"])

print("Latest WT value is",existing_data["WT"][-1])
print("Latest KO value is",existing_data["KO"][-1])
```

Exercise 2

Babraham Bioinformatics

Iterators, Loops and Conditionals

Babraham Bioinformatics

Iteration over a list (or tuple, or set)

```
animals = ["dog","cat","mouse","elephant"]

for animal in animals:                  # Note the colon at the end.

    print(animal.upper())               # You can use the loop name
                                        # here. It steps through all
                                        # the values



for animal in reversed(animals): # Note NOT animals.reverse()
    print(animal.upper())
```

Code blocks in python

- Many operations define a 'code block'
  - Code to run within a loop
  - Code to run (or not) depending on a logical test

- Code blocks in python are defined by indentation - the amount of space at the start of your lines
  - Correct formatting is required for the code to work properly
  - Blocks can be spaces or tabs of any number (at least 1)
  - Python style recommends 4 spaces for indentation
    - A good editor will handle all of this for you, just use tab to insert indentation

Indenting code blocks

```
animals = ["dog","cat","mouse","elephant"]

for animal in animals:
    print(animal.lower())
    print(animal.upper())

print("Finished listing animals")
```

Block starts

Block finished

4 space indent

There must be at least 1 statement in a block. You can use `pass` as a way to make a dummy statement if you really need to.

Iterators

- When iterating over large, dynamically created, data sets it would be inefficient to make the whole set just to iterate over it
  - The reversed version of a large list
  - All of the numbers between 1 and 1,000,000,000
- Python uses 'iterators' as a memory efficient way to handle this
- Need to use `list()` if you really want the whole set

Iterators

```
animals = ["dog","cat","mouse","elephant"]

reversed(animals)
# <list_reverseiterator object at 0x0000020CAB011220>

list(reversed(animals))
# ['elephant', 'mouse', 'cat', 'dog']
```

Ranges

- Simple and efficient way to loop over sets of integers

```
for i in range(5):              # 0, 1, 2, 3, 4
    print(i)

for i in range(5,10):           # 5, 6, 7, 8, 9
    print(i)

for i in range(5,16,2):         # 5, 7, 9, 11, 13, 15
    print(i)

for i in range(16,5,-2):        # 16, 14, 12, 10, 8, 6
    print(i)
```

Iterating over list indices (and values)

```
animals = ["dog","cat","mouse","elephant"]
```

```
for i in range(len(animals)):
    print(i, animals[i])
```

```
0 dog
1 cat
2 mouse
3 elephant
```

```
for i in enumerate(animals):
    print(i[0])
    print(i[1])
```

```
for i,animal in enumerate(animals):
    print(i, animal)
```

- `enumerate` makes an iterator of tuples (index, value) over a list
- `i,animal = (1,"dog")` is an easy way to extract tuple or list values into separate variables

Iterating over dictionaries

- Two options
  1. Iterate over the keys, and then use them to look up the values
  2. Iterate simultaneously over the keys and values

```
animal_dict = {"elephant":"big", "dog":"medium", "mouse":"small"}

for animal in animal_dict.keys():       # keys() iterates just the keys
    print(animal, animal_dict[animal])


for animal,size in animal_dict.items():# items() gives a (key,value) tuple
    print(animal, size)
```

Conditional Tests

- A way to have a block of code which runs under some circumstances but not others.
- Consists of a logical test followed by a code block which executes only if the test is 'true'
- Code blocks are indented in the same way that loop blocks were

What is 'true'

- Logical tests evaluate code to be 'true' or 'false', so what's true? Actually easier to say what's false.
  - The logical `False` value
  - The `None` value
  - Empty lists, tuples and dictionaries
  - Any numerical zero value (`int` or `float`)

- Everything else is true

Constructing a logical test

```
test = False
another_test = True

if test:
    print("Test was true")

elif another_test:
    print("Test was False, but Another Test was true")

else:
    print("Neither test nor another test were true")
```

Optional

In the interactive interpreter you can't have a blank line between `if` / `elif` / `else`. In a script you can.

Logical test operators

Operation	Meaning	Example
`<`	strictly less than	`5 < 6.5`
`<=`	less than or equal	`4 <= 4`
`>`	strictly greater than	`2.3 > 1.6`
`>=`	greater than or equal	`4.001 >= 4`
`==`	equal	`"simon" == "simon"`
`!=`	not equal	`4 != 4`
`is`	object identity	`[1,2] is [1,2]`
`is not`	negated object identity	`[1,2] is not [1,2]`

Equality: == vs `is`

- == tests whether the two sides are equivalent
- `is` tests whether the two sides are the same
- For simple scalars (numbers, text) always use ==
  - Don't use == to compare float (fractional) numbers
- For more complicated structures either might be relevant

```
"simon" == "SIMON"  # False - case sensitive

10 == 10            # True - OK as they're integers

[1,2,3] == [1,2,3]  # True - data is equivalent

[1,2,3] is [1,2,3]  # False - not the same list
```

Logical tests on data structures

```
animals = ["dog","cat","mouse","elephant"]
animal_dict = {"elephant":"big", "dog":"medium", "mouse":"small"}

if "gerbil" in animals:                  # Works for lists, sets, tuples
    print("There's a gerbil there")
else:
    print("Sorry, no gerbils")

if "elephant" in animal_dict:            # Tests against the keys
    print("We know about elephants")

if "medium" in animal_dict.values(): # Tests against the values
    print("There's a medium animal")
```

Compound Tests

- You can use the operators and / or to link logical tests or not to negate a test

```
if "gerbil" not in animals:
    print("No gerbils here")


for animal,size in animal_dict.items():
    print(animal)
    if size=="big" and animal.startswith("e"):
        print("I bet it's an elephant")
```

`while` loops

- A way to execute a code block repeatedly until a logical test becomes false
- The logical test uses the same code as if statements
- Generally something within the loop needs to change the data used in the logical test
  - Otherwise the loop will run forever (an infinite loop)
  - There are ways to break out of a loop

while loops

```
I guessed 1
I guessed 4
I guessed 7
I guessed 1
I guessed 5
I stopped guessing
```


```python
#!python

import random

guessed_number = -1

while guessed_number != 5:
    guessed_number = random.Random().randint(0,10)
    print("I guessed",guessed_number)

print("I stopped guessing")
```

```python
#!python
data = []

while True:             # An infinite loop
    answer = input("Enter data: ")

    if answer.strip() == "":
        break

    if not answer.isnumeric():      # Tests for integer
        print("Sorry, wasn't a number")
        continue

    data.append(int(answer))

mean = 0

for i in data:
    mean += i           # Shortcut for mean = mean + i

mean /= len(data)

print("The mean of",len(data),"observations was",mean)
```

Using `continue` and `break`

```python
#!python

data = {}
sample_count = 0

while sample_count < 10:
    sample_count += 1
    print("\n\nMeasurement",sample_count)
    sample = input("Sample Name: ")
    value = float(input("Data Value: "))

    if not sample in data:
        data[sample] = []

    data[sample].append(value)

for sample in data.keys():
    data[sample].sort()
    print("Sample",sample,"had",len(data[sample]),"measures:",data[sample])
```

Exercise 3

Babraham Bioinformatics

String Processing

Babraham Bioinformatics

Creating Strings

```
text_single = 'simple single quotes'  # No real difference
                                      # between single and quotes
text_double = "simple double quotes"  # double quotes

text_escaped = 'It\'s tricky writing apostrophes'

text_special = 'header1\theader2\ndata1\tdata2\n' # Newlines / tabs

text_multi = """I can write
over several
lines
"""
```

Testing Strings

- `isalnum()` Are all characters in the string alpha-numeric
- `isalpha()` Are all characters in the string alphabetic
- `isascii()` Are all the characters standard ASCII
- `isdecimal()`
- `isdigit()` Test for numbers (plus varying extended characters)
- `isnumeric()`
- `isidentifier()` Is the text a reserved word in python
- `islower()` Is it lowercase
- `isupper()` Is it uppercase
- `istitle()` Is it title case (initial capital)
- isprintable() All characters are printable (not carriage returns etc)
- `isspace()` All characters are spaces/tabs

Splitting and Joining

- Convert between lists/tuples and delimited strings

```
text_delim = "Jan_Feb_Mar_Apr_May"

months = text_delim.split("_")                  # [Jan Feb Mar Apr May]

new_delim = ":".join(months)                    # Jan:Feb:Mar:Apr:May

",".join(["one",2,"three",4])                   # Fails, can't join int
",".join(["one",str(2),"three",str(4)])         # Works, but ugly!
```

Strings as tuples

- Behind the scenes strings are stored as a tuple of letters
- You can use similar methods to lists/tuples on strings

```
big = "arabidopsis"

small = big[-6:]                    # dopsis
small = big[::-1]                   # sispodibara

if "bido" in big:
    print("Found substring")        # Works
```

String Operators

- Strings can be 'added' or 'multiplied'

```
text1 = "join"
text2 = 'me'

text_joined = text1 + text2           # joinme
text_joined = "can't" + "mix" + 100   # Fails, can't add int
text_joined = "can" + "mix" + str(100)# Works

text_multi = text1 * 4                # joinjoinjoinjoin
```

Building strings with data

```
#!python

sample = "WT"
count = 23
total_count = 101

print("Sample",sample,"comprised",count/total_count,"of all measures")

message = "Sample "+sample+" comprised "+str(count/total_count)+" of all measures"

# Sample WT comprised 0.2277227227722773 of all measures
```

Format Strings (f-strings)

- String concatenation with + is not elegant, and fails with anything which isn't a string
- Format strings are a more flexible way of mixing text and data
- Works with all data types without conversion
- Only available since python 3.7
- Regular strings, prepended by f and with { } expressions in them

Format Strings (f-strings)

```
year = 1983
name = "Simon"
kids = ["Fred","Ethel"]

print(f"{name} was born in {year} and has {kids}")

# Simon was born in 1983 and has ['Fred', 'Ethel']

print(f"{name} was born in {year} and has {' & '.join(kids)}")

# Simon was born in 1983 and has Fred & Ethel
```

Number formatting in f-strings

- `{data:[align][width][delimiter].[precision]}`
  - Align is < (left) > (right) ^ (center)
  - Width is number of characters
  - Delimiter is 1000s separator (normally `,` or _)
  - Precision is number+letter
    - `f` is fixed decimal places
    - `g` is significant figures
- Examples
  - `{data:<20.2f}` Occupy 20 spaces, align left show 2 decimal places
  - `{data:,.3g}` Take what space you need. Add commas. Show 3 sig figs

Number formatting in f-strings

```
fnum = 19876.12345

print(f"Simple={fnum}")                 # Simple=19876.12345

print(f"Decimal Places={fnum:.2f}")     # Decimal Places=19876.12

print(f"SigFigs={fnum:.3g}")            # SigFigs=1.99e+04

print(f"Commify={fnum:,.0f}")           # Commify=19,876

print(f"FixWidthR='{fnum:>15}'")        # FixWidthR='    19876.12345'

print(f"FixWidthC='{fnum:^15}'")        # FixWidthC='  19876.12345  '
```

Complex Matching

- Simple literal string matching can be achieved using either `in` or methods such as `index` or `find`
- More complex, ambiguous patterns can be found using methods from the `re` (regular expression) package - part of the standard library
- Regular expressions are used in many languages and are the same in all of them.

Common methods from `re`

- `re.findall` Find all matches to a pattern. Return a list of hits
- `re.search` Find the first match to a pattern. Return a hit object
- `re.finditer` Find all matches to a pattern. Return a hit iterator
- `re.split` Like `str.split` but using a pattern not literal text
- `re.sub` Find and replace based on a pattern

Constructing Patterns

- Patterns are strings, but containing special characters
- Special characters allow for ambiguity in the pattern

`.`	Anything
`[ade]`	Set of allowed characters
`|`	Either/or. Surrounded by () if ambiguous
`*`	Zero or more
`+`	One or more
`?`	None or one
`{}`	Specific number of occurrences, exact or {min,max}
`^`	Starts with
`$`	Ends with
`()`	Capture group, used to capture part of a match for later use
`\`	Way to escape a special character you want to use literally

Pattern Examples

`b.b`	b [anything] b
`^ga*t`	starts with g, then any number of a then t
`tata+$`	tat then one or more a at the end of the string
`c[aeiou]{4}`	c then exactly 4 vowels
`\.txt$`	.txt at the end of a string
`lane([0-9])\.fq`	lane then a captured number then .fq
`file\.(fq|fastq)`	File dot fq or fastq

Character group shortcuts in regular expressions

- Certain groups of characters are so common there is a shortcut

`\d`	Digits (0-9)
`\D`	Non-digits
`\s`	Any whitespace (spaces or tabs)
`\S`	Any non-whitespace
`\w`	Any word character (letters, numbers and underscore)
`\W`	Any non-word character

- Eg: `lane([0-9])\.fq` could be `lane(\d)\.fq`

Finding matches

```
import re

text="From sample 1535 we counted 712 colonies on 12 plates"

hits = re.findall("\d+",text)
print(hits)                                  # ['1535','712','12'] - always strings

no_hits = re.findall("bacteria",text)
print(no_hits)                               # [] Empty list

if re.findall("\d+",text):                   # Works because a populated list is
    print("There were numbers")              # true but and empty list is false

if not re.findall("bacteria",text):
    print("No bateria here")
```

Capturing parts of matches

- Using the search function allows you to use capture groups
  - Any part of the regex surrounded in round brackets
  - Can have multiple captures in the same regex

```
import re

text="From sample 1535 we counted 712 colonies on 12 plates"

hits = re.search("(\d+) colonies.* (\d+) plates",text)
```

  - Sample number and plate number will be captured

Capturing parts of matches

- From the hit object which is returned by `re.search` or `re.finditer`
  - `span()` is the position of the whole match, 2 element tuple, start, end
  - `groups()` is a tuple of data in the capture groups

```
import re

text="From sample 1535 we counted 712 colonies on 12 plates"

hits = re.search("(\d+) colonies.* (\d+) plates",text)

if hits is not None: # Tests whether there was a match
    print(f"Matched between {hits.span()[0]} and {hits.span()[1]}")
    print(f"Colonies={hits.groups()[0]} Plates={hits.groups()[1]}")

# Matched between 28 and 53
# Colonies=712 Plates=12
```

Find and Replace

- Use the `re.sub` function to replace a match
  - Regex for what to match
  - Replace with a string

```
import re

text="From sample 1535 we counted 712 colonies on 12 plates"

new_text = re.sub("sample \d+","sample 1234", text)
print(new_text)

# From sample 1234 we counted 712 colonies on 12 plates
```

```python
#!python
import re

hisat_text = """HISAT2 summary stats:
        Total reads: 77188721
                Aligned 0 time: 8862035 (11.48%)
                Aligned 1 time: 60127229 (77.90%)
                Aligned >1 times: 8199457 (10.62%)
        Overall alignment rate: 88.52%
"""
stats = {}
lines = hisat_text.split("\n")

for line in lines:
    if line.isspace():
        continue

    if "Total reads" in line:
        stats["total"] = line.split(":")[1]

    elif line.strip().startswith("Aligned"):
        hits = re.search("Aligned\s+(\S+)\s+times?:\s+(\d+)",line)

        if hits is None:
            print(f"Couldn't match expected pattern in {line}")
            continue

        stats[hits.groups()[0]] = hits.groups()[1]

for stat in stats.keys():
    print(f"{stat} had value {stats[stat]}")
```

Exercise 4

Babraham Bioinformatics

Reading and Writing Files

Babraham Bioinformatics

Constructing File Paths

- A File paths is a string of folder / file name separated by a delimiter (usually /)
- On windows you often see \ used to separate path elements, but as part of a regular python string this needs to be escaped so appears as /
  - `"c:/Users/andrewss/python/example.py"`
  - `"c:/Users/andrewss/python/example.py"`
- Traditionally a mix of `os`, `os.path`, `glob` and `shutils` packages were used to deal with paths
- These have largely been supplanted by the `pathlib` package

Using `pathlib`

- The `pathlib` package defines the `Path` data type

- Once you create a `Path` you can call methods on it
  - Constructing / joining paths
  - Testing paths
  - Listing files / folders
  - Creating / deleting files or folders

Using File Paths

- Construct a starting path using `Path("location")`
  - Can be a string with a full path
  - Can be a set of path segments (drives, folders, files)

```
from pathlib import Path

singlepath = Path("C:/Users/andrewss/Desktop")
multipath = Path("C:/","Program Files","Python39","python.exe")
```

Joining File Paths

- Start from a base path and add a file name to the end
- Can use / as a shortcut for `joinpath`

```
from pathlib import Path

base_path = Path("C:/Users/andrewss/Desktop")

final_path = base_path.joinpath("data.txt")
# C:\Users\andrewss\Desktop\data.txt

final_path = base_path / "data.txt"
# Also works
```

Path sections

C:/Program Files/Python39/python.exe

- p.anchor = "C:/"
- p.drive = "C:"
- p.parent = Path('C:/Program Files/Python39')
- p.name   = "python.exe"
- p.stem   = "python"
- p.suffix = ".exe"
- str(p)   = "C:/Program Files/Python39/python.exe"

Useful `Path` methods

- `p.exists()` — Does this path exist
- `p.is_file()` — Is this a file (not a dir)
- `p.is_dir()` — Is this a directory (not a file)

- `p.stat()` — Get statistics about the path
  - `p.stat().st_size` — The size (in bytes)
  - `p.stat().st_mtime` — When it was last modified (epoch seconds)
  - `p.stat().st_atime` — When it was last accessed (epoch seconds)*

*Not guaranteed to work on every filesystem

Reading Text Files

- The standard process for reading a text file is:

1. Construct the path to the file
2. Check the path exists
3. Open a 'stream' to the file - a variable from which data can be read
4. Read the data line by line in a loop
5. Close the stream

Full Read Example

```
from pathlib import Path
import sys

base_path = Path("C:/Users/andrewss/Introduction to Python/Python Intro Data")
file_path = base_path.joinpath("babraham_citations.txt")

if not file_path.exists():
    print(f"Couldn't find {file_path}", file=sys.stderr)
    sys.exit(1)

file_stream = open(file_path, encoding="UTF-8")

for line in file_stream:
    line = line.strip()
    if "Nat" in line:
        print(line)

file_stream.close()
```

Check file

Report problem

Create a stream

Read from stream

Close stream

Simpler reads using `with`

```
from pathlib import Path
import sys

base_path = Path("C:/Users/andrewss/Introduction to Python/Python Intro Data")

with open(base_path/"babraham_citations.txt", encoding="UTF-8") as file_stream:

    for line in file_stream:
        line = line.strip()
        if "Nat" in line:
            print(line)
```

Clarifications: Text File Encodings

```
with open(file_path, encoding="UTF-8") as file_stream:
```

```
UnicodeDecodeError: 'charmap' codec can't decode byte 0x81 in position 4326:
character maps to <undefined>
```

- Text files are just files of numeric values decoded into symbols
- Original text file encoding was ASCII
  - Numbers 0 to 127 represent common letters/numbers/symbols
  - ASCII can't represent many characters eg © ã α etc.
  - Need more than one number per character
  - Several different schemes, 'Latin-1', 'cp1252' etc.
  - UTF-8 is now taking over and should be used
  - On OSX and Linux UTF-8 is the default encoding, but not windows

Error Reporting

```
if not file_path.exists():
    print(f"Couldn't find {file_path}", file=sys.stderr)
    sys.exit(1)
```

- All OSs have two types of output
  - `sys.stdout` Standard output for expected output
  - `sys.stderr` Standard error for errors, warnings or progress

- You can exit your program early using `sys.exit()`
  - The exit value should be 0 if the program exited normally
  - Non zero exit means there was a problem (error)

Using Exceptions (Errors)

- Exceptions are a more robust way of reporting and dealing with problems
- They will construct messages and code traces to allow debugging
- Exceptions can be 'caught' so you can deal with them internally
- There is a generic Exception but also more specific ones

```
+-- Exception
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- NameError
      +-- OSError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- TimeoutError
      +-- SystemError
      +-- TypeError
      +-- ValueError
```

Using Exceptions (Errors)

```
if not file_path.exists():
    raise FileNotFoundError(f"Couldn't find {file_path}")
```

```
try:
    with open(file_name, encoding="UTF-8") as file_stream:

        for line in file_stream:
            line = line.strip()
            if "Nat" in line:
                print(line)

except Exception as ex:
    print("Oops it went wrong, never mind")
    print(ex)
```

Writing to text files

```
out_path = base_path.joinpath("interesting_genes.txt")

with open(out_path,"w") as out:
    out.write("nanog")              # write doesn't automatically
    out.write("\n")                 # add a newline.

    print("brca2", file=out)        # print does
```

- File open modes
  - r = read, also rt (read text) is sometimes used (default)
  - rb = read binary (ie non-text)
  - w = write as text (and delete any existing content)
  - wb = write binary (and delete any existing content)
  - a = append to existing content

File Reading Packages

- `csv` - parses comma separated value files
- `gzip` - reads gzip compressed data
- zipfile - read data from zip files
- tarfile - read data from tar files
- `pysam` - reads SAM or BAM files (not in standard library)
- openpyxl - reads xlsx/xslm Excel files (not in standard library)

Listing Files

- Simple
  - Use the `iterdir` method of a directory Path
- Filtered
  - Use the `glob` method with a pattern containing a `*` (eg `*.txt`)
- Recursive
  - Use the `rglob` method instead of `glob`

```python
#!python
from pathlib import Path

base = Path("C:/Users/andrewss/git")

for d in base.iterdir():
    if d.is_dir():
        print(f"Found repository {d.name}")


for f in (base/"aws_training_images").glob("*.sh"):
    print(f"Found shell script {f.name}")


html_count = 0

for _ in base.rglob("*html"):
    html_count += 1

print(f"Found {html_count} HTML files")
```

Creating Directories

- Create a Path to a location which doesn't exist
- Call the `mkdir` method
  - Set `parents=True` if you want to create several directories

```
new_path = Path("C:/Users/andrewss/Data/Output/")

if not new_path.exists():
    new_path.mkdir(parents=True)
```

Deleting Files

- The `unlink` method of `Path` will remove files or empty directories
  - Be careful - files are not recycled, just deleted
- It won't recursively delete directories and data
  - You can use shutil.rmtree for this if you're **really** sure

```python
#!python
from pathlib import Path
import gzip
import re

base_path = Path.home()/"Desktop"/"Introduction to Python"/"Python Intro Data"
polya_lengths = {}

with gzip.open(base_path/"example.fq.gz", mode="rt", encoding="UTF-8") as fq:
    for line in fq:
        if line.startswith("@SRR"):
            sequence = fq.readline()
            max_a = 0
            for polya in re.findall("A+",sequence):
                if len(polya) > max_a:
                    max_a = len(polya)

            if not max_a in polya_lengths:
                polya_lengths[max_a] = 0

            polya_lengths[max_a] += 1


with open(base_path/"palengths.txt","w") as out:
    for palength in sorted(polya_lengths.keys()):
        print(f"{palength}\t{polya_lengths[palength]}", file=out)
```

Exercise 5

Babraham Bioinformatics

Writing Functions and Larger Scripts

Babraham Bioinformatics

Better Code Structure

- When your scripts get larger
  - Split the code into modular chunks (functions)
  - Share code between scripts
  - Add some documentation
  - Parse command line options
  - Write tests

- Functions help with
  - Code readability
  - Code maintainability and testing
  - Code reuse

Writing functions

```
def calculate_gc(sequence):
    total = len(sequence)
    gc = sequence.count("G") + sequence.count("C")

    percent = 100*(gc/total)

    return percent

seq = "GATTCGATAGCTAG"
gc = calculate_gc(seq)
print(f"The GC content of {seq} is {gc:.1f}")
# The GC content of GATTCGATAGCTAG is 42.9
```

Functions are processed in order

```
seq = "GATTCGATAGCTAG"
gc = calculate_gc(seq)
print(f"The GC content of {seq} is {gc:.1f}")


def calculate_gc(sequence):
    total = len(sequence)
    gc = sequence.count("G") + sequence.count("C")

    percent = 100*(gc/total)

    return percent
```

```
Traceback (most recent call last):
  File "c:\Users\andrewss\Intro_Python\functions.py", line 3, in <module>
    gc = calculate_gc(seq)
NameError: name 'calculate_gc' is not defined
```

Putting everything into a function

```
def main():
    seq = "GATTCGATAGCTAG"
    gc = calculate_gc(seq)
    print(f"The GC content of {seq} is {gc:.1f}")


def calculate_gc(sequence):
    total = len(sequence)
    gc = sequence.count("G") + sequence.count("C")

    percent = 100*(gc/total)

    return percent

main()
```

Scripts can be packages too

script.py

```
import sequtils

def main():
    seq = "GATTCGATAGCTAG"
    gplusc = sequtils.calculate_gc(seq)
    print(f"The GC content of {seq} is {gplusc:.1f}")


main()
```

sequtils.py

```
def calculate_gc(sequence):
    total = len(sequence)
    gc = sequence.count("G") + sequence.count("C")

    percent = 100*(gc/total)

    return percent
```

Am I a script, or am I a package?

script2.py

```
import script1

gc = script1.calculate_gc("GGG")
print(f"Script2 calculated {gc}")
```

```
The GC content of GATGCTAG is 50.0
Script2 calculated 100.0
```

Being able to simply reuse functions from other scripts is great, but how do we stop the 'script' part of script1 from running when it's being used as a package?

script1.py

```
def main():
    seq = "GATGCTAG"
    gplusc = calculate_gc(seq)
    print(f"The GC content of {seq} is {gplusc:.1f}")

def calculate_gc(sequence):
    total = len(sequence)
    gc = sequence.count("G") + sequence.count("C")

    percent = 100*(gc/total)
    return percent

main()
```

The __name__ special variable

- Variables surrounded by double underscores are designed for mostly internal use, and are created automatically. Sometimes they are useful to use directly.
  - When a script is executed directly then __name__ has a value of "__main__"
  - When a script is executed because it's called by being imported into another script __name__ is set to the script name
  - We can change our behaviour depending on the value of __name__

Standard Script Structure

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

def main():
    pass

def myfunction():
    pass

def myotherfunction():
    pass

if __name__ == "__main__":
    main()
```

Code for this scripts direct functionality goes in here

These functions can be used from main or can be used in other scripts if this file has been imported into them.

The `main()` function only runs when the script is directly executed

Adding Documentation

- Simple function documentation can be added as a string immediately below the function definition

```
def calculate_gc(sequence):
    """Calculates the GC content of an
    uppercase sequence string"""
    total = len(sequence)
    gc = sequence.count("G") + sequence.count("C")

    percent = 100*(gc/total)

    return percent
```

```
>>> import sequtils
>>> help(sequtils.calculate_gc)
Help on function calculate_gc in module
sequtils:

calculate_gc(sequence)
    Calculates the GC content of an
    uppercase sequence string
```

Encapsulation and Scoping

- Functions should be self contained
  - They shouldn't rely on the presence of variables outside the function
  - They should only send data back via a return statement
  - It's OK to create new variables within the function but these won't affect the global environment

- It is possible to affect a global variable in a function, but it requires extra code

Encapsulation and Scoping

```
message = "Original value"

def changeme():
    new_message = "Changed message"
    message = new_message
    print(f"Inside, message is {message}")


print(f"Outside, message was {message}")
changeme()
print(f"Outside, message is {message}")
```

```
Outside, message was Original value
Inside, message is Changed message
Outside, message is Original value
```

If we try to use message at the start of the changeme function we'd get:

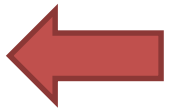
```
UnboundLocalError: local variable
'message' referenced before
assignment
```

Accessing global variables

- Global variables are generally a bad idea and you should minimise their use
  - Variables scoped within the `main()` function and passed to other functions as needed are preferred
- There are times they can be useful though
- To access them in a function you need to use the global keyword

Accessing global variables

```
message = "Original value"

def changeme():
    global message
    print(f"Inside, message was {message}")
    new_message = "Changed message"
    message = new_message
    print(f"Inside, message is {message}")


print(f"Outside, message was {message}")
changeme()
print(f"Outside, message is {message}")
```



Says we're importing the message variable from the global environment

```
Outside, message was Original value
Inside, message was Original value
Inside, message is Changed message
Outside, message is Changed message
```

Command Line Options

- You can make your script more flexible by using command line arguments to change options, or the data to process

```
myscript.py cutoff=20 file=data1.csv
```

- Anything written after the script is put into a list accessed via `sys.argv`

Using sys.argv directly

```
import sys
def main():
    for i,v in enumerate(sys.argv):
        print(f"{i} was {v}")

if __name__=="__main__":
    main()
```

```
>python argv.py cutoff=20 data=data1.csv
0 was argv.py
1 was cutoff=20
2 was data=data1.csv
```

More robust command lines with argparse

```
import argparse

def main():
    options = parse_arguments()

    print(f"Cutoff is {options.cutoff} data file is {options.data}")

def parse_arguments():
    parser = argparse.ArgumentParser(description="Analyse my data")

    parser.add_argument("--cutoff", help="The cutoff to use for the analysis", default=20, type=int)
    parser.add_argument("data", help="The data file to process", type=str)

    return parser.parse_args()

if __name__ == "__main__":
    main()
```

More robust command lines with argparse

```
usage: commandline.py [-h] [--cutoff CUTOFF] data

Analyse my data

positional arguments:
  data                  The data file to process

optional arguments:
  -h, --help            show this help message and exit
  --cutoff CUTOFF       The cutoff to use for the analysis
```

More robust command lines with argparse

```
commandline.py somefile.csv
Cutoff is 20 data file is somefile.csv


commandline.py --cutoff 26 test.csv
Cutoff is 26 data file is test.csv


commandline.py --cutoff=26
usage: commandline.py [-h] [--cutoff CUTOFF] data
commandline.py: error: the following arguments are required: data


commandline.py --cutoff=no test.csv
usage: commandline.py [-h] [--cutoff CUTOFF] data
commandline.py: error: argument --cutoff: invalid int value: 'no'
```

Testing your code

- Adding tests to your code is a good way to ensure the functionality you're developing is working correctly
- You don't need anything else, but the `pytest` framework makes running tests somewhat easier
- Some people advocate 'test driven development', basically you write the tests first, and then write code until all of the tests pass

The `pytest` framework

- Not part of the standard library, so need to install with pip
- Looks for files called `test_*.py` or `*_test.py`
- Runs `test_` functions within these files containing `assert` statements
  - Asserts are statements containing a test, which produce an exception if the test result is not `True`
- Reports on the success of the tests

```python
def calculate_gc(sequence):
    """Calculates the GC content of an
    uppercase sequence string"""
    total = len(sequence)
    gc = sequence.count("G") + sequence.count("C")

    percent = 100*(gc/total)

    return percent

def reverse_complement(sequence):
    """Calculates the reverse complement of
    an uppercase sequence string"""

    rev = sequence[::-1]
    revcomp = rev.translate(
        str.maketrans("GATC","CTAG")
    )

    return revcomp
```

```python
import sequtils

def test_gc():
    seq = "GATC"
    assert(
        sequtils.calculate_gc(seq)==50
    )


def test_revcomp():
    seq = "GGAT"
    assert(
        sequtils.reverse_complement(seq)=="ATCC"
    )
```

```
python -m pytest
======================== test session starts ========================
platform win32 -- Python 3.9.1, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: C:\Users\andrewss\Desktop\Introduction to Python
collected 2 items

test_sequtils.py ..                                            [100%]

========================= 2 passed in 0.06s =========================
```

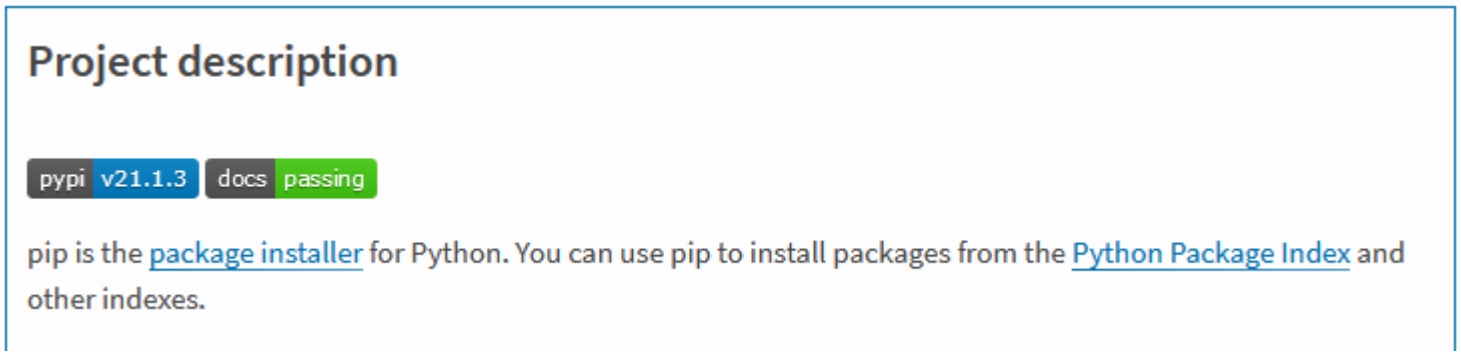
Exercise 6

Babraham Bioinformatics

Using external resources

Babraham Bioinformatics

Installing Additional Packages



sys.path

```
>>> import sys
>>> sys.path
['', 'C:/Program Files/Python39/python39.zip', 'C:/Program Files/Python39/DLLs',
'C:/Program Files/Python39/lib', 'C:/Program Files/Python39',
'C:/Users/andrewss/AppData/Roaming/Python/Python39/site-packages',
'C:/Users/andrewss/AppData/Roaming/Python/Python39/site-packages/win32',
'C:/Users/andrewss/AppData/Roaming/Python/Python39/site-packages/win32/lib',
'C:/Users/andrewss/AppData/Roaming/Python/Python39/site-packages/Pythonwin',
'C:/Program Files/Python39/lib/site-packages']
```

- Packages are search for in the order of `sys.path`
- Stops at the first hit
- Some will be admin only, others are user-writeable

Installing with pip

```
pip install …
python -m pip install …

pip install biopython
pip install --user biopython
pip install --upgrade biopython
pip uninstall biopython
```

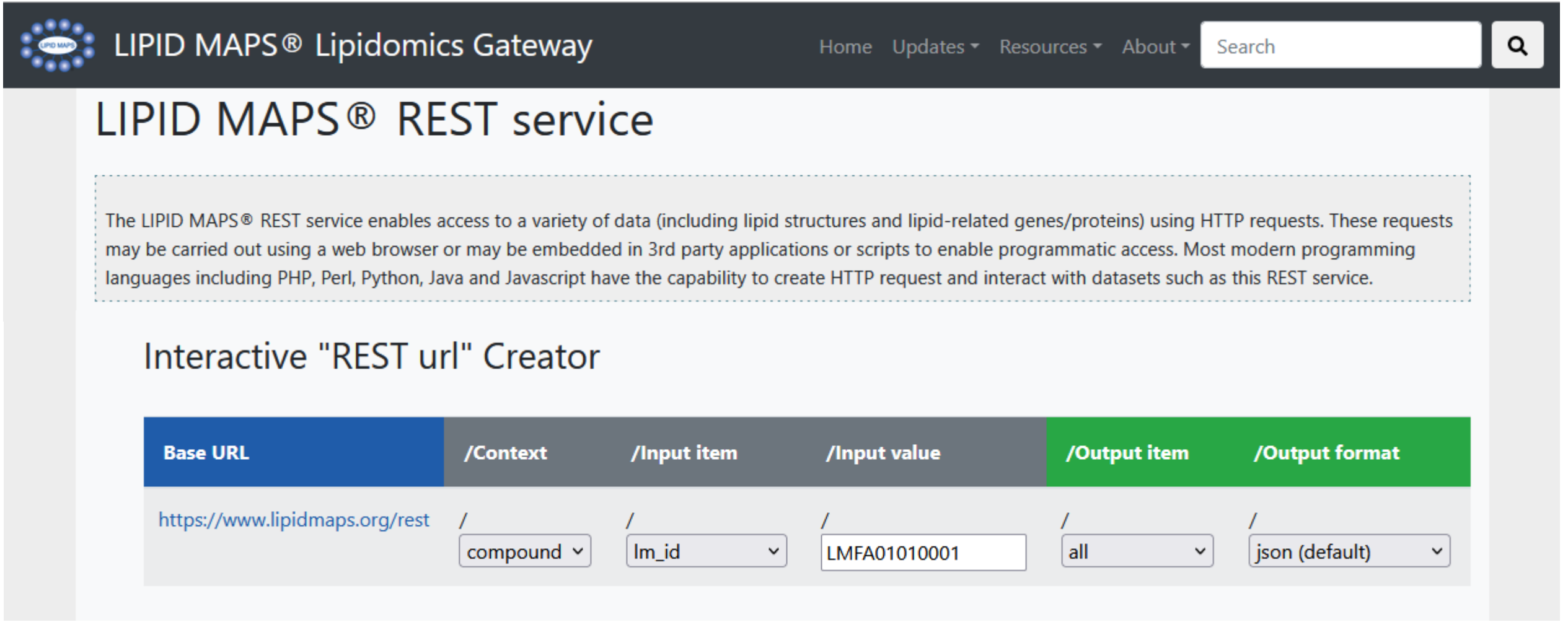
Virtual Environments

```
pip install --user venv

python -m venv mynewproject

source mynewproject/bin/activate  [Linux/Mac]
.\mynewproject\Scripts\activate   [Windows]

deactivate
```

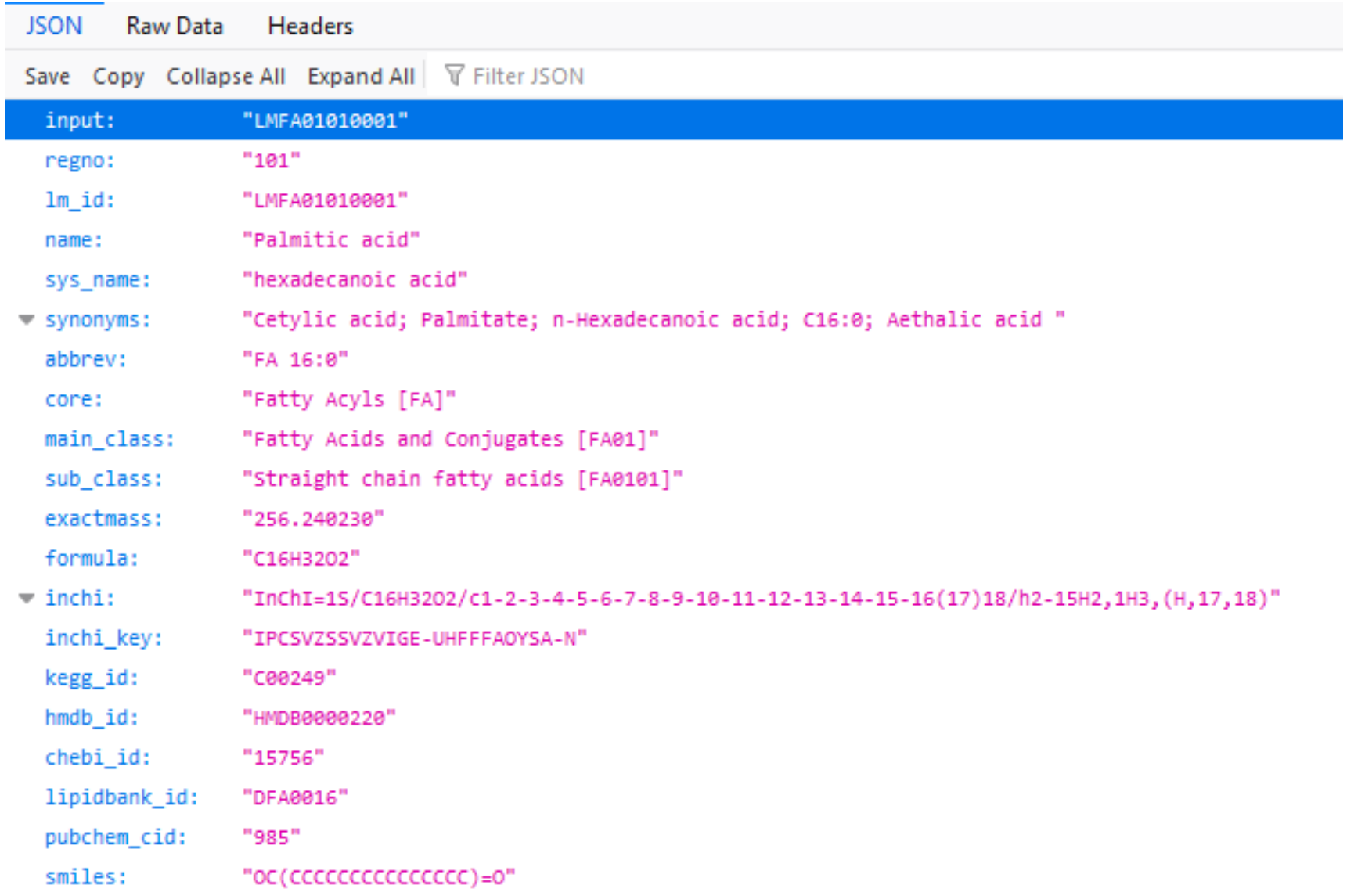
Getting data from REST APIs

- Many data sources offer a simple way to pull information from an online resource, called a REST API
- These are accessed by a structured URL defining the data required
- Data is normally returned in JSON format which can be easily parsed by python

REST Example



https://www.lipidmaps.org/rest/compound/lm_id/LMFA01010001/all/json



JSON - JavaScript Object Notation

- Simple text format
- Composed of lists and dictionaries
- Easily transposed into equivalent python data structures

- The `json` package is part of the standard library
  - Create json from list/dictionary `json.dumps`
  - Create list/dictionary from text json `json.loads`

Reading web data using `requests`

- Convenient package for reading data from the web
- Supports HTTP HTTPS and FTP URLs

```
import requests

def main():
    lmid= "LMFA01010001"
    json_data = fetch_lm_json(lmid)

    print(f"LMID {lmid} is {json_data['name']} and has mass {float(json_data['exactmass']):.2f}")


def fetch_lm_json(lmid):
    answer = requests.get(f"https://www.lipidmaps.org/rest/compound/lm_id/{lmid}/all/json")
    return answer.json()

if __name__=="__main__":
    main()
```

More complex requests

g:Profiler

g:Profiler API

g:Profiler requests are generally made as POST requests with a JSON body and they return JSON output.

g:GOSt

URL: https://biit.cs.ut.ee/gprofiler/api/gost/profile/
METHOD: POST
PARAMETERS:

organism

ID of species to be queried. List of possible ID-s can be seen at the organisms list page.

```
organism:"hsapiens"
```

query

List of genes to be queried. Can be a list of strings or a dictionary of lists if multiple queries are submitted simultaneously.

```
query:["CASQ2", "CASQ1", "GSTO1", "DMD", "GSTM2"]
```

```
query:{
                first_query:["CASQ2", "CASQ1", "GSTO1", "DMD", "GSTM2"],
                second_query:["MLXIPL","SMARCB1","PIH1D1","SMARCA4","AGER"]
                }
```

g:GOSt query result fields

These are the result fields for most simple queries.

name
Term name.

description
Term description if available. If not available, repeats the term name.

native
Term ID in its native namespace. For non-GO terms, the ID is prefixed with the datasource abbreviation.

parents
List of native IDs that are hierarchically above the term. For non-hierarchical datasources, points to artificial root node if applicable.

p_value
Hypergeometric p-value after correction for multiple testing.

goshv
Internal g:Profiler numeric ID. Unique for the term. Not consistent across data updates.

significant
Indicator for statistically significant results.

https://biit.cs.ut.ee/gprofiler/page/apis

More complex requests

```
#!/usr/bin/env python3
import requests

def main():
    genes = "ENSG00000007171,ENSG00000141367,etc".split(",")
    request = {
        "organism": "hsapiens",
        "query": genes,
        "sources":[],
        "user_threshold":0.01
    }

    result = requests.post("https://biit.cs.ut.ee/gprofiler/api/gost/profile/", json=request)

    for hit in result.json()["result"]:
        print(f"{hit['name']}\t{hit['p_value']:.3f}")

if __name__ == "__main__":
    main()
```

More complex results

```
>python.exe gprofiler.py

clathrin binding        0.000
clathrin-coated vesicle 0.000
clathrin-coated vesicle membrane        0.000
coated vesicle  0.000
regulation of leukocyte mediated immunity       0.000
Formation of annular gap junctions      0.000
coated vesicle membrane 0.000
Gap junction degradation        0.000
leukocyte mediated cytotoxicity 0.000
clathrin-coated endocytic vesicle       0.000
regulation of immune system process     0.001
vesicle 0.001
receptor-mediated endocytosis   0.002
regulation of leukocyte mediated cytotoxicity   0.002
```

Running external programs from python

- The `subprocess` package provides the `Popen` and `run` functions which have options for the most common variations
  - Launch a process and wait for it to complete and check the exit code
  - Launch a process and collect output from STDOUT or STDERR
  - Launch a process and forget about it (later check exit if you like)

Main `subprocess` options

- Arguments
  - Can be a single string of program plus arguments, need `shell=True` for this
  - Can be a list of separate command components, doesn't need shell
- Check (for `subprocess.run`)
  - `check=False` (default) returns a result which you can query to see if it worked
  - `check=True` raises an Exception if the process exits in an error state
- Output
  - `stdout` and `stderr` can either be left to print to the screen, or sent to a pipe where they can be read like a file
  - If you're reading text from `stdout/stderr` then set `encoding` so it appears as text

```python
import subprocess

# Start a process and wait for it to complete
print("Starting notepad")
exit_status = subprocess.run("c:/Windows/system32/notepad.exe")
print(f"Notepad finished {exit_status}")


# Start a process and forget about it
print("Starting forgotten notepad")
subprocess.Popen("c:/Windows/system32/notepad.exe")
print(f"Notepad running")


# Start a process and collect data from it
print("Starting collected process")
with subprocess.Popen(
    "c:/Windows/system32/ipconfig.exe",
    stdout=subprocess.PIPE,
    encoding="UTF-8"
    ) as running_proc:

    for line in running_proc.stdout:
        print(f"Found line {line.strip()}")
```

Start and wait
Check exit code

Start and forget
No exit check

Start and collect stdout output as a stream

Use `with` so you don't have to manually clean up process or streams

Additional little tricks
(if we have time…)

Babraham Bioinformatics

List Comprehension

- A useful shortcut for performing the same operation on all members of a list

```
data = [1,2,3,4,5]
print("\t".join(data))
```

```
 print("\t".join(data))
TypeError: sequence item 0: expected str instance, int found
```

```
data = [1,2,3,4,5]

fixed_data = []
for d in data:
    fixed_data.append(str(d))

print("\t".join(fixed_data))
```

```
1	2	3	4	5
```

List Comprehension

```
data = [1,2,3,4,5]
fixed_data = [str(x) for x in data]
print("\t".join(fixed_data))
print("\t".join( [str(x) for x in data] ))
```

```
# Filtering
print("\t".join(  [str(x) for x in data if x>3]  ))

# Conditional Transformation
print("\t".join(  ["odd" if x%2 else "even" for x in data]  ))
```

Debugging

- Python has a built in debugger which you can use to help sort out problems in your code

- You can start the debugger at any point in your code by inserting a call to the `breakpoint()` function (python 3.7+)
  - Good for logic errors

- You can enter the debugger instead of crashing by running
  `python3 -m pdb crashing_program.py`
  - Good for tracing the cause of crashes

Debugging

```
def print_lines(file):

    line_number = 1

    for line in file:
        line_number += 1
        if line_number == 10:
            breakpoint()
            break

        print(line)

    print("Finished")
```

```
Line 1
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Finished
```

```
Line 7
Line 8
> c:\users\andrewss\debugger.py(12)print_lines()
-> break
(Pdb)
```

Debugger commands

- Print the value of an expression (often just a variable)

```
(Pdb) p line
'Line 9'
```

- Step to the next line of code

```
(Pdb) s
> c:\users\andrewss\debugger.py(16)print_lines()
-> print("Finished")
```

- Allow the program to continue to the next breakpoint (or end)

```
(Pdb) c
Finished
```