

# Programming Principles and Practice using Python

Ade Fewings, Aaron Owen

Bangor University

Updated: 2022-02-16

# eResearch

- Supercomputing Wales
  - Available to researchers at Bangor
- Research Software Engineers
- Collate expert knowledge into an open and shared centralised repository
  - Yammer
  - Github
  - Workshops
  - Projects
  - Acknowledgements

# Training Workshops

- Introduction to the Linux Shell
- Version Control Using Git
- **Programming Principles and Practice using Python**
- Advanced Python
- Parallel Processing in Python
- Machine Learning with Python\*

See and discuss on the Yammer group.

Suggestions for new training welcome.

# Resources

## Online Tutorials

- [Software Carpentry - Programming with Python](#)
- [Python.org - The Python Tutorial](#)
- [Google's Python Class](#)
- [Introduction to Python](#)
- [Real Python Tutorials](#)

# Environment

## Cross Platform

- [Google Colaboratory](#)
- [Online Python compiler](#), [Online Python IDE](#), and [online Python REPL](#)
- [python.org shell](#)

## Python at Bangor

- Visit <https://jupyter.bangor.ac.uk/jupyter/hub/login>
- Login using your Bangor University credentials.
- Click the `Python 3.8` notebook.

# What is Python

- Interpreted, interactive and object-oriented programming language.
- Highly scalable - Small applications to large-scale enterprise solutions.
  - Desktop, Scientific libraries and Web applications (MySCW).
- There are two versions of Python: 2.x and 3.x.
  - Prefer 3.x for new development.
  - If the codebase you are working on uses 2.x, then use 2.x.
    - Many third-party libraries use 2.x.
  - Python 2.7 will not be maintained past 2020.
- Many developers will test ideas and prototypes in Python and drop down to a lower level language if required.
- <https://docs.python.org/3/>

# Workshop

- Using Python as a vehicle to learn good programming practises.
- Reflect on the concepts.
- Feel free to ask questions.

# Python Shell

- For those using an online platform a Python Shell is ready to use.
- For those using the `compute` node, type `python3` and an interactive Python Shell will open.

```
[issa16@compute ~]$ python3
Python 3.6.8 (default, Aug  7 2019, 17:28:10)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-39)] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

- Enter a line of code, hit enter and python immediately executes the statement.

```
name='Aaron'
```

```
name or print(name)
```

- To exit the Python Shell type `exit()` and hit enter.
  - Will fail on repl.it.



# Built-In Data Types

- Data Types are an important concept in any programming language.
- Variables can store data of different data types and different data types behave differently.
- Different data types have different memory requirements.
  - Text Type: `str`
  - Numeric Types: `int`, `float`, `complex`
  - Sequence Types: `list`, `tuple`, `range`
  - Mapping Types: `dict`
  - Set Types: `set`, `frozenset`
  - Boolean Types: `bool`
  - Binary Types: `bytes`, `bytearray`, `memoryview`

# Variables

- Variables are containers for data.
- Syntax to declare a variable is `name = value`
  - `name` is assigned the `value`
- Can name a variable anything you like apart from a few **reserved keywords**.
- In Python, a variable can be assigned a value of one type and then later re-assigned a value of a different type. Personally, I think its best to keep the type the same throughout it's lifetime.
- Try not to use cryptic names
  - `i` as a for loop index is fine, `i` as a non index variable can be cryptic
  - `n= 'Aaron'` or `name= 'Aaron'` ?
- Clear and self-explanatory variable names improve code readability for you and others.

# Naming

- **Camel Case:** Second and subsequent words are capitalized, to make word boundaries easier to see.
  - Example: `numberOfCollegeGraduates`
- **Pascal Case:** Identical to Camel Case, except the first word is also capitalized.
  - Example: `NumberOfCollegeGraduates`
- **Snake Case:** Words are separated by underscores.
  - Example: `number_of_college_graduates`
- Is it a matter of personal preference for your projects.
- Be consistent.
- Often a style will be defined for open source projects.
- Personal preference is to use snake case for variables and functions and use Pascal Case for class names.

# String (str)

- Strings are sequences of character data.
- String may be delimited using either a single or double quote.

```
name='Aaron'  
name  
  
name="Beth"  
name  
  
type(name)  
  
print("This string contains a single quote (') character.")  
print('This string contains a double quote (") character.')  
print('''This string has a single (') and a double (") quote.''' )  
  
print("""This is a  
string that spans  
across several lines""")
```

- String Methods

# Integer (int)

- Python interprets a sequence of decimal digits without any prefix to be a decimal number (base 10).
- Effectively no limit on how long an integer value can be. It is constrained by the amount of memory your system has.

a=1

a

b=123456

b

```
type(b)
```

## # Advanced

c=0b10     # Binary 2

$d = 0 \times 10$  # Hexadecimal 16

e=0o10 # Octal 8

```
print(123123123123123123123123123123123123123123123123123123123 + 1)
```

123123123123123123123123123123123123123123123124

# Floating-Point Numbers (float)

- The `float` type in Python designates a floating-point number.
- `float` values are specified with a decimal point.
- Optionally, the character `e` or `E` followed by a positive or negative integer may be appended to specify scientific notation.

```
a=4.2
```

```
a
```

```
b=.2
```

```
b
```

```
c=.4e7
```

```
c
```

```
d=4.2e-4
```

```
d
```

```
type(d)
```

# Booleans

- Variables of Boolean type have one of two values, True or False.

```
a=True
```

```
a
```

```
b=False
```

```
b
```

```
type(b)
```

- Expressions in Python are often evaluated in a Boolean context.
- Non-Boolean objects can be evaluated in a Boolean context.

# Check Variable Data Type

- Often it can be useful to check the data type of a variable.

```
a=1
type(a)  # int

b=2.2
type(b)  # float

c=False
type(c)  # bool
```



# Comments

- When writing code, it's important to make sure that your code can be easily understood by others.
- Giving variables good names, defining explicit functions, and organising your code are all great ways to do this.
- Additionally, we can use comments to increase the readability of the code.
- Single single comment

```
a=1 # This is a comment and will not run
```

- Multiline comment

```
"""  
This is a  
multiline comment  
and will not run  
"""
```

- Don't Repeat Yourself, code smell (over commenting) and rude comments.

# Scripts

- Entering commands into the Python Shell interactively is good for learning and quick testing of code and libraries.
- Eventually, you will develop more complex programs and need to run a piece of code repeatedly.
- A python script is a reusable set of code. It is essentially a file that contains a sequence of Python instructions.
- Python scripts are plain text, so you can edit them in any text editor.
- **repl.it** provides a file (main.py) and calls `python3 main.py` when you click run.
- **Google Colab** provides a notebook that can be saved to your Google drive.
- Demo

# Scripts

- Running scripts from the command line.
- Demo
  - Create a new file called `main.py`

```
def main():  
    print('Called main()')  
  
if __name__ == '__main__': # Entry point  
    main()
```

```
python3 main.py
```

- Demo without `__name__`

# Data structures

There are four collection data types in the Python programming language:

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.
- When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

# Lists

- A list is a collection which is ordered and changeable.
- In Python, lists are written with square brackets.

```
names = ["Aaron", "Beth", "George"]  
print(names)
```

```
['Aaron', 'Beth', 'George']
```

# Lists (Access Items)

- You access the list items by referring to the index number:

```
names = ["Aaron", "Beth", "George"]  
print(names[0])  
print(names[1])  
print(names[2])  
print(names[3]) # ???
```

Aaron

Beth

George

```
Traceback (most recent call last): File "main.py", line 5, in  
<module>     print(names[3]) # ???  
IndexError: list index out of range
```

# Lists (Negative Indexing)

- Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item etc.

```
names = ["Aaron", "Beth", "George"]  
print(names[-1])  
print(names[-2])
```

```
George  
Beth
```

# Lists (Add an item)

- To add an item to the end of the list, use the `append()` method:

```
names = ["Aaron", "Beth", "George"]  
names.append("Dereck")  
print(names)
```

```
['Aaron', 'Beth', 'George', 'Dereck']
```



# Lists (Add an item at specified index)

To add an item at the specified index, use the `insert()` method:

```
names = ["Aaron", "Beth", "George"]  
names.insert(1, "Mia")  
print(names)
```

```
['Aaron', 'Mia', 'Beth', 'George']
```

# Lists (Find an item)

- The `index()` method returns the index in the list of the first item whose value is equal to x.

```
names = ["Aaron", "Beth", "George", "George"]  
x = "Beth"  
idx = names.index(x)  
print(f'Found {x} at the following index: {idx}')
```

```
Found Beth at the following index: 1
```

# Lists (Find an item)

- To find all occurrences of items whose value is equal to x, we can use a `for` loop and the `enumerate` function.
- More on this in later in the slides.

```
names = ["Aaron", "Beth", "George", "George"]
x = "George"
idxs = []
for idx, val in enumerate(names):
    if val == x:
        idxs.append(idx)
print(f'Found {x} at the following indexes: {idxs}')
```

```
Found George at the following indexes: [2, 3]
```

## Advanced

```
names = ["Aaron", "Beth", "George", "George"]
x = "George"
idxs = [idx for idx, val in enumerate(names) if val == x]
print(f'Found {x} at the following indexes: {idxs}')
```

# Lists (Remove an item)

- The `remove()` method removes the specified item:

```
names = ["Aaron", "Beth", "George"]  
names.remove('Aaron')  
print(names)
```

```
['Beth', 'George']
```

# Lists (Clear a list)

- The `clear()` method empties the list:

```
names = ["Aaron", "Beth", "George"]  
names.clear()  
print(names)
```

```
[]
```

# Tuples

- A tuple is a collection which is ordered and unchangeable. In Python, tuples are written with round brackets.

```
names = ("Aaron", "Beth", "George")  
print(names)
```

```
('Aaron', 'Beth', 'George')
```

- Behaves the same as a list apart from the immutability.
- Technically you could convert a tuple to a list, change the values and convert back to a tuple.

# Dictionaries

- A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

```
person = {  
    "name": "Aaron",  
    "age": 32,  
    "title": "Mr"  
}  
print(person)
```

```
{'name': 'Aaron', 'age': 32, 'title': 'Mr'}
```

# Dictionaries (Accessing Items)

- You can access the items of a dictionary by referring to its key name, inside square brackets:

```
person = {  
    "name": "Aaron",  
    "age": 32,  
    "title": "Mr"  
}  
print(person["name"])  
print(person.get("name"))  
print(person.get("x", "default if x does not exist"))
```

Aaron

Aaron

default **if** x does not exist



# Dictionaries (Change Values)

- You can change the value of a specific item by referring to its key name:

```
person = {  
    "name": "Aaron",  
    "age": 32,  
    "title": "Mr"  
}  
person["name"]="Beth"  
person["age"]=29  
person["title"]="Miss"  
  
print(person)
```

```
{'name': 'Beth', 'age': 29, 'title': 'Miss'}
```

# Dictionaries (Check if Key Exists)

- To determine if a specified key is present in a dictionary, use the `in` keyword:

```
person = {  
    "name": "Aaron",  
    "age": 32,  
    "title": "Mr"  
}  
  
if "name" in person:  
    print("Yes, 'name' is one of the keys in the person dictionary")  
else:  
    print("No, 'name' is not one of the keys in the person dictionary")
```

Yes, 'name' is one of the keys **in** the person dictionary

# Dictionaries (Adding Items)

- Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
person = {  
    "name": "Aaron",  
    "age": 32,  
    "title": "Mr"  
}  
person["job"]="Research Software Engineer"  
print(person)
```

```
{'name': 'Aaron', 'age': 31, 'title': 'Mr', 'job': 'Research Software Engineer'}
```

# Dictionaries (Removing Items)

- The `pop()` method removes the item with the specified key name:

```
person = {  
    "name": "Aaron",  
    "age": 32,  
    "title": "Mr"  
}  
person.pop("age")  
print(person)
```

```
{'name': 'Aaron', 'title': 'Mr'}
```

# Dictionaries (Nested)

- A dictionary can also contain many dictionaries, this is called nested dictionaries
- Can be an elegant solution when used correctly

```
pets = {  
    "mia": {  
        "age": 4,  
    },  
    "boy cat" : {  
        "age": 9  
    },  
    "girl cat" : {  
        "age": 9  
    }  
}  
  
print(pets)  
print(pets["mia"])  
print(pets["mia"]["age"])
```

```
{'mia': {'age': 4}, 'boy cat': {'age': 9}, 'girl cat': {'age': 9}}  
{'age': 4}  
4
```

# List of Dictionaries

- Lists can store any data type, including dictionaries.

```
participants = []

print(participants)

person_1 = {
    "name" : "Aaron",
    "age" : 32,
}

person_2 = {
    "name" : "Beth",
    "age" : 29,
}

participants.append(person_1)
participants.append(person_2)

print(participants)
```

# Conditional Statements

- Python supports the usual logical conditions from mathematics:
  - Equals: `a == b`
  - Not Equals: `a != b`
  - Less than: `a < b`
  - Less than or equal to: `a ≤ b`
  - Greater than: `a > b`
  - Greater than or equal to: `a ≥ b`
  - These conditions can be used in several ways, most commonly in "if statements" and loops.

# if

- An "if statement" is written by using the if keyword.

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

b is greater than a

- In this example we use two variables, `a` and `b`, which are used as part of the `if` statement to test whether `b` is greater than `a`.



# Elif

- The elif keyword is python's way of saying "if the previous conditions were not true, then try this condition".

```
a = 33
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
```

a and b are equal

- In this example, `a` is equal to `b`, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

# Else

- The `else` keyword catches anything which isn't caught by the preceding conditions.

```
a = 200
b = 33
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

a is greater than b

- In this example `a` is greater than `b`, so the first condition is not true, also the `elif` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".

# And

- The `and` keyword is a logical operator, `and` is used to combine conditional statements
- Test if `a` is greater than `b`, AND if `c` is greater than `a`

```
a = 200
b = 33
c = 500
if a > b and c > a:
    print("Both conditions are True")
```

Both conditions are True

# Or

- The `or` keyword is a logical operator, `or` is used to combine conditional statements
- Test if `a` is greater than `b`, OR if `a` is greater than `c`:

```
a = 200
b = 33
c = 500
if a > b or a > c:
    print("At least one of the conditions is True")
```

```
At least one of the conditions is True
```

# For Loops

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- With the for loop, we can execute a set of statements, once for each item in a list, tuple, set etc.
- Print each fruit in a fruit list

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)
```

- Loop through a string

```
for x in "banana":  
    print(x)
```

# For Loops (break)

- With the break statement we can stop the loop before it has looped through all the items:

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    print(x)  
    if x == "banana":  
        break
```

```
apple  
banana
```

# For Loops (continue)

- With the continue statement, we can stop the current iteration of the loop, and continue with the next

```
fruits = ["apple", "banana", "cherry"]  
for x in fruits:  
    if x == "banana":  
        continue  
    print(x)
```

```
apple  
cherry
```

# For Loops (range)

- To loop through a set of code a specified number of times, we can use the range() function.
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):  
    print(x)
```

```
0  
1  
2  
3  
4  
5
```



# For Loops (range)

- If you are displaying values in a loop, you might use `end=` to cause the values to be displayed on one line, rather than on individual lines:

```
for n in range(10):  
    print(n, end=(' ' if n < 9 else '\n'))
```

```
0 1 2 3 4 5 6 7 8 9
```

# For Loops (Else in For Loop)

- The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished.
- Print all numbers from 0 to 5, and print a message when the loop has ended.

```
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

```
0  
1  
2  
3  
4  
5  
Finally finished!
```

# For Loops

- Loop through a dictionary

```
person = {  
    "name": "Aaron",  
    "age": 32,  
    "title": "Mr"  
}  
  
for x, y in person.items():  
    print(x, y)
```

```
name Aaron  
age 31  
title Mr
```

# Functions

- So far we have been using Python interactively.
- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

# Creating a Function

- In Python a function is defined using the `def` keyword.
- To call a function, use the function name followed by parenthesis.

```
def test_function():  
    print("Hello from test_function")  
  
test_function()
```

# Parameters

- Information can be passed to functions as a parameter.
- Parameters are specified after the function name, inside the parentheses.
- You can add as many parameters as you want, just separate them with a comma.
- Avoid more than five. Function should be broke up into smaller parts.

```
def test_function(name):  
    print(name + " Owen")  
  
test_function("Aaron")  
test_function("Beth")  
test_function("George")
```

```
Aaron Owen  
Beth Owen  
George Owen
```

# Default Parameter Value

- The following example shows how to use a default parameter value.
- If we call the function without parameter, it uses the default value.

```
def test_function(country = "Norway"):  
    print("I am from " + country)
```

```
test_function("Sweden")  
test_function("India")  
test_function()  
test_function("Brazil")
```

```
I am from Sweden  
I am from India  
I am from Norway  
I am from Brazil
```

- Avoids `None` or `''` checks

# Passing a List as a Parameter

- You can send any data types of a parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

```
fruits = ["apple", "banana", "cherry"]
```

```
def test_function(food):  
    for x in food:  
        print(x)
```

```
test_function(fruits)
```

```
apple  
banana  
cherry
```



# Return Values

- To let a function return a value, use the `return` statement.

```
def test_function(x):  
    return(5 * x)
```

```
print(test_function(3))  
print(test_function(5))  
print(test_function(9))
```

```
15  
25  
45
```

# Keyword Arguments (kwargs)

- You can also send arguments with the `key = value` syntax.
- This way the order of the arguments does not matter.

```
def test_function(child3, child2, child1):  
    print("The youngest child is " + child3)  
  
test_function(child1 = "George", child2 = "Amelia", child3 = "Linus")
```

The youngest child is Linus

# Arbitrary Arguments

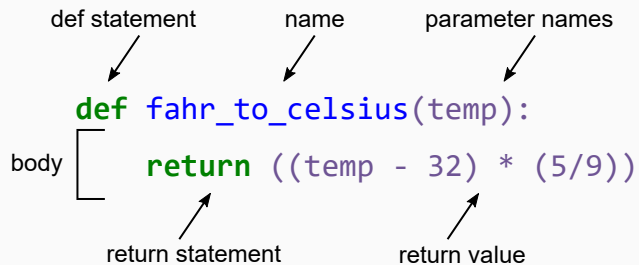
- If you do not know how many arguments that will be passed into your function, add a `*` before the parameter name in the function definition.
- This way the function will receive a tuple of arguments and can access the items accordingly

```
def test_function(*kids):  
    print("The youngest child is " + kids[2])  
  
test_function("George", "Amelia", "Linus")
```

The youngest child is Linus

# Example (Fahrenheit to Celsius)

```
def fahr_to_celsius(temp):  
    return ((temp - 32) * (5/9))
```



The diagram illustrates the components of the function definition `def fahr_to_celsius(temp):` and its body `return ((temp - 32) * (5/9))`. Labels with arrows point to specific parts of the code:

- def statement**: Points to the `def` keyword.
- name**: Points to the function name `fahr_to_celsius`.
- parameter names**: Points to the parameter `temp` in the parentheses.
- body**: A bracket on the left side of the function body, pointing to the `return` statement.
- return statement**: Points to the `return` keyword.
- return value**: Points to the expression `((temp - 32) * (5/9))`.

- Opens with `def` keyword followed by the name of the function.
- Parenthesized list of parameter names.
- Body of the function.
- Concludes with the `return` keyword followed by the return value.

# Example (Fahrenheit to Celsius)

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

```
Freezing point of water: 0.0 C  
Boiling point of water: 100.0 C
```

# Nested Functions

- It is possible to call a function from within another function.
- Try to avoid too much inner function calling, can be difficult to debug.

```
def func_1():  
    print('func_1 called')  
    func_2()  
  
def func_2():  
    print('func_2 called')  
  
func_1()
```

```
func_1 called  
func_2 called
```

# Advice

- Functions should do one thing and do it well.
- As we will see with TDD, a function that does too many operations and mutates too much data can be problematic.
- Follow the same naming rules as mentioned for variables.

# Documentation

- Documentation is very important.
- Often it can be the only form of communication between software developers.
- In Python, functions are often given a 'docstring' to assist users of the function.

```
def func_1(some_arg):  
    '''  
    This is the docstring for func_1  
  
    Examples  
    -----  
    >>> func_1(22)  
    22  
    '''  
    print(some_arg)
```

```
help(func_1)
```



# Read data from files

- Create a text file named `data.txt` and enter your name and age on separate lines.

Aaron

32

- Open the `data.txt` file using the built-in `open()` function that Python provides.

```
data=[]

with open('data.txt', 'r') as data_file:
    for line in data_file:
        line = line.strip() # Remove newline character
        print(f"Add '{line}' to the data list")
        data.append(line)

print(data)
```

More on this in the Advanced Python workshop

# Write data to files

- Write a list of data items to a file names `name-list.txt`

```
data=['Aaron','Beth', 'George']

with open('name-list.txt', 'w+') as data_file:
    for item in data:
        data_file.write(item + '\n')

print(data)
```

More on this in the Advanced Python workshop

# Read, Update, Write

- Open a file named `planets.txt`.
- For each listed planet, reverse the name.
- Write the reversed planet names to `planets-reversed.txt`.

```
filename_input = 'planets.txt'
filename_output = 'planets-reversed.txt'

data=[]

# Read data from the input file
with open(filename_input, 'r') as data_file:
    for line in data_file:
        line = line.strip() # Remove newline character
        data.append(line)

# For each data item, reverse the string
for i in range(len(data)):
    data[i]=''.join(reversed(data[i]))

# Write data to the output file
with open(filename_output, 'w+') as data_file:
    for item in data:
        data_file.write(item + '\n')
```

# Errors

- All programmers encounter errors, from beginner to expert.
- Knowing how to deal with errors can drastically reduce the amount of time you debug your code and avoid the feeling that coding is a hopeless endeavour.
- In Python, errors have a specific form called a traceback.

```
ice_creams = ["chocolate", "vanilla", "strawberry"]  
print(ice_creams[3])
```

```
Traceback (most recent call last):  File "main.py", line 2, in <module>  
    print(ice_creams[3])  
IndexError: list index out of range
```

- <https://docs.python.org/3/library/exceptions.html>

# Syntax Errors

- Python can not figure out how to read your program
- Often developers forget a colon at the end of a line, accidentally add one space too many when indenting or forget a parenthesis.

```
def some_function()  
    msg = 'Hello Aaron'  
    print(msg)  
    return msg
```

```
File "main.py", line 1      def some_function()  
                             ^
```

SyntaxError: invalid syntax

# Variable Name Errors

- A common type of error is the `NameError`.
- Occurs when you try to use a variable that does not exist.

```
print(x)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'x' is not defined

# Index Errors

- If you try to access an item in a list or a string that does not exist, then you will get an error.

```
letters = ['a', 'b', 'c']
print("Letter #1 is", letters[0])
print("Letter #2 is", letters[1])
print("Letter #3 is", letters[2])
print("Letter #4 is", letters[3])    # ??
```

Letter #1 is a

Letter #2 is b

Letter #3 is c

Traceback (most recent call last):

File "main.py", line 5, in <module>

print("Letter #4 is", letters[3])

IndexError: list index out of range

# File Errors

- If you try to read a file that does not exist, you will receive a `FileNotFoundError`.
- If you attempt to write to a file that was opened read-only, Python 3 returns an `UnsupportedOperationError`.
- Examples on File IO after TDD.

```
file_handle = open('myfile.txt', 'r')
```

```
Traceback (most recent call last): File "main.py", line 1, in <module>
    file_handle = open('myfile.txt', 'r')
FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'
```



# Key Points

- Tracebacks can look intimidating, but they give us a lot of useful information about what went wrong in our program, including where the error occurred and what type of error it was.
- An error having to do with the 'grammar' or syntax of the program is called a `SyntaxError`. If the issue has to do with how the code is indented, then it will be called an `IndentationError`.
- A `NameError` will occur when trying to use a variable that does not exist. Possible causes are that a variable definition is missing, a variable reference differs from its definition in spelling or capitalisation, or the code contains a string that is missing quotes around it.
- Containers like lists and strings will generate errors if you try to access items in them that do not exist. This type of error is called an `IndexError`.
- Trying to read a file that does not exist will give you an `FileNotFoundError`. Trying to read a file that is open for writing, or writing to a file that is open for reading, will give you an `IOError`.

# Defensive Programming

- How can I make my programs more reliable?
  - Assertions
  - Test-Driven Development

## Goal

- Write programs that check their operation.
- Write and run tests for widely-used functions.
- Make sure we know what "correct" actually means.
- Reduce the amount of time we spend in debug mode.

# Assertions

- As developers, we need to assume that mistakes will happen and we need to put measures in place to guard against them.
- This is known as defensive programming.
- The most common way to be defensive is to add **assertions** to our code.
- When Python reads an assertion
  - It evaluates to assertions condition
  - If True, nothing happens
  - If False, program halts and prints an error

# Assertion Demo

Ensure only positive numbers are processed

Code

```
numbers = [0.1, 1.5, -2.4, 4.8]
total = 0
for i in numbers:
    assert i > 0, 'Data should only contain positive values'
    total += i

print(f'total is {total}')
```

Result

```
Traceback (most recent call last):  File "main.py", line 4, in <module>
    assert i > 0, 'Data should only contain positive values'
AssertionError: Data should only contain positive values
```

# Assertions

- Fall into three categories
  - Precondition
    - A condition must be true at the start of a function for it to work correctly. Example, checking input args.
  - Postcondition
    - A condition that the function guarantees is true when it finishes. Example, checking result is valid.
  - Invariant
    - A condition that is always true at a particular point inside a piece of code. Example, age can't be negative.

# Pre and Post Conditions Demo

- Suppose you are writing a function called `average` that calculates the average numbers in a list. What pre and post conditions would you check for?

*# A possible pre-condition:*

```
assert len(input_list) > 0, 'List length must be non-zero'
```

*# A possible post-condition:*

```
assert min(input_list) ≤ average ≤ max(input_list),  
'Average should be between min and max of input values (inclusive)'
```

# Assertions

- They are not just about catching errors.
- They help people to understand your code - documentation.
- Whenever you fix a bug or error in your code, write an assertion that catches the mistake.

# Test-Driven Development

- An assertion checks that something is true at a particular point in the program.
- We need a method to check the overall behaviour of a piece of code.
- Test-Driven Development (TDD)
  - Write the tests before writing the function
  - If people write tests after writing the code to be tested, they are subject to confirmation bias. They subconsciously write tests to show that their code is correct, rather than find errors.
  - Helps to write modular code and view dependencies.



# Key Points

- Program defensively, i.e., assume that errors are going to arise and write code to detect them when they do.
- Put assertions in programs to check their state as they run, and to help readers understand how those programs are supposed to work.
- Use preconditions to check that the inputs to a function are safe to use.
- Use postconditions to check that the output from a function is safe to use.
- Write tests before writing code to help determine exactly what that code is supposed to do.

# Test-Driven Development Demo

hello\_world.py

```
def get_greetings():  
    return 'Hello World!'
```

simple\_unit\_tests.py

```
import unittest  
from helloworld import get_greetings  
  
class HelloworldTests(unittest.TestCase):  
  
    def test_get_helloworld(self):  
        self.assertEqual(get_greetings(), 'Hello World!')  
  
if __name__ == '__main__':  
    unittest.main()
```

python3 simple\_unit\_tests.py -v

