# Coding Fundamentals

## DevOps Level 4: Module 2

# CONTENTS

## Python Programming Basics

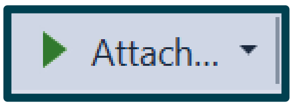
### Visual Studio

For this course, we would like you to use Microsoft Visual Studio as your IDE. This tool is a great development environment and has excellent debugging and testing facilities.

There are some great online IDEs you can use as well, such as repl.it.

Printing to console window

You can either use the play button to run a program or you can press the F5 key.



### Making Comments in Code

When making complex programs, files can become quite large. Readability is an important part of Python programming, and comments help us with this.

If you use the # key in Python, anything after the # on the same line will be ignored. For example,

```
# python will ignore this

print('Hello World!') # display greetings
```

### Data Types

A data type is an attribute of data that tells us how a programmer intends to use the data. Data types define the operations that you can carry out on the data, the meaning of the data, and the way values of that type can be stored.

### Numbers

There are two main types of numbers in Python: integers and floating points (floats). An integer is a whole number (1, 2, 3) and a float is a decimal number (1.23, 0.005).

### Strings

A string is a block of text - a word, sentence or paragraph.

### Boolean

A boolean (or 'bool' for short) can only store two possible values: true or false.

### Variable Names

When naming variables in Python, we use letters, numbers and underscores.

salay$ = 1500 ✗ y_city='London' ✓

It is important to not use already reserved words such as 'print'.

Remember, Python is case sensitive, so Age and age are two different identifiers in Python.

## User Input
Sometimes, we want the user of the program to enter some data for the program to handle. We do this with the input function:

```
name = input("prompt")
```

This will ask the user to type a piece of data and Python will store the entry in the variable identifier. You can customise the prompt the user gets with the string passed in to the function.

The user's entry is always a string.

## Concatenation
You can attach strings to each other using the + operator.

```
username= 'bob'
print('Hello' + username) -> "Hellobob"
print('Hello ' + username) -> "Hello bob"
```

## Casting
You can convert data types using casting. For example:

```
int("2") -> 2
str(3) -> "3"
float(3) -> 3.0
```

We do this with user input if we wish for them to enter a number.

```
age = int(input('What is your age? '))
```

# Control Flow

A programme's control flow is the order in which the code is executed.

In Python, If, Elif and Else statements are used to determine the control flow.

## If

If statements are used to run a block of code if the condition is met.

You can chain **if-else** statements to build more complex conditional statements.

```
is_dev_cool = False
if is_dev_cool:
  print("Dev is cool")
else:
  print("Dev isn't cool")
```

## Operators

Sometimes we need relational operators to specify more complex conditions. These include:

- Greater than >
- Greater than or equal to >=
- Less than <
- Less than or equal to <=
- Equal to ==
- Not equal to !=

```
X = 5
if x == 5:
  print("X equals 5")
else:
  print("X does not equal 5")
```

## Elif

Elif is short for Else If and this allows for chains of If statements.

```
salary = 2500
if salary > 100000:
   print('Band A')
elif salary > 55000:
   print('Band B')
elif salary > 32000:
   print('Band C')
elif salary > 25000:
   print('Band D')
else:
   print('Band E')
```

## OR and AND

You can use OR and AND to combine multiple comparisons to logical operators.

```
course='Python'
age = 19

if course == 'Python' and age > 18:
   print('Welcome!')
```

## While Loops

Loops are another way to change the control flow. They help us with repeating sections of code for a specified number of times.

A while loop will run the same piece of code while the statement is true.

```
x = 1
 while x < 5:
    print(x,"Hello World")
    x = x + 1
```

In this loop, we are printing “Hello World” while x is less than 5, and then adding 1 to x. So eventually, x will be equal to 5 and the while loop will stop. This program will then, in full, print “Hello world” four times.

## Break

Use the break statement to end any kind of loop early:

```
total = 0
answer = 'y'

while answer == 'y':
   total += int(input('Enter a number (1-10)'))

   if total >= 21:
      break

   answer = input('Get another number?')

print('Total is ',total)
```

This while loop will allow the user to add numbers to the total but, if the total goes above 21, then the loop will stop.

## For Loops

For loops allow you to iterate through a list, string or range.

```
names = [“ben”, “harry”, “dev”]

for name in names:
        print(name)
```

This program will iterate through the list of names and then print each name.

Notice that we define a new variable ‘name’ that represents that current element of the list.

## Range

Range is a useful function normally used with for loops. It defines the range between any two numbers, which a for loop can then iterate through.

```
range(5) -> 0,1,2,3,4
range(3,5) -> 3,4
```

### With for loops

```
for x in range(5):
   print(x, "Hello World")
```

This program, much like the while loop above, will print “Hello World” five times.

## Lists

Lists refer to multiple values which are all contained in, and accessible through, a single variable. You can edit the values in a list the list has been created.

When working with lists, we reference the position of an item in the list. However, it’s important to note that indexing in lists starts counting from 0.

```
names = [“ben”, “harry”, “dev”]

print(names[1]) -> “harry”
```

We can get a group of elements in a list using list slicing:

```
names = [“ben”, “harry”, “dev”]
print(names[1:3])->[“ben”,”harry”]
```

### Len

The len() function is used to get the length of a list or string:

```
numbers = [1,3,5,7]

print(len(numbers)) -> 4
```

A string can be treated as a list of characters, which is why you can iterate through a string and also use len to find the length of a string.

### Appending

You can add new elements to a list using append:

```
numbers = [1,3,5,7,9]

numbers.append(88)

print(numbers) -> [1,2,3,5,7,9,88]
```

### Removing

You can remove elements from a list using remove:

```
numbers = [1,3,5,7,9]

numbers.remove(3)

print(numbers) -> [1,2,5,7,9]
```

Note: if a list has duplicate entries, remove will remove all the duplicates.

### Del
You can also use del to remove a specific index of the list:

```
del(numbers[0])
```

This will remove the first element in the list.

### In
You can use the IN operator to check if the list contains a specific element:

```
big = [1,2,3,4,5]
small = 1
if small in big:
        print("Yes!")
```

### List Functions
Lists come with many built-in functions. We are going to take a look at two of the most commonly used: sort and split.

### Sort
Sort will put the list in a certain order:

```
numbers = [4,2,8,5,3]
numbers.sort()
print(numbers) -> [2,3,4,5,8]
```

### Split
Split allows us to split a string into a list based on a given separator:

```
numbers = "1,2,3,4,5"
number_list=numbers.split(",")
print(number_list) -> [1,2,3,4,5]
```

## Functions

A function is a block of code in Python written to perform one task, but it will perform this task many times as you need it to. You call a function by typing the name of a function and then parentheses after the name. For example:

```
int()
```

### Parameters

Sometimes, functions need parameters to work. For example, the int function above needs to take in a string or float that can be turned into an integer:

```
int(“3”)
```

### Return values

A function can use a value like int() to return the new integer data type. This can be stored in a variable for later use. But a function doesn’t always have to return a value.

### In-built

Python has many built-in functions, which are functions that are automatically accessible to any user who has installed Python, such as print, str, int, range etc.

Lists and strings have built-in functions to help with formatting. We’ve explored a couple of these already; sort and split. These are called by referencing the data being formatted:

```
numbers = [3,1,2]

numbers.sort()
```

### Library

Some functions aren’t installed automatically, but you can install and use them through a library. For example, math is a library that has the functions sqrt, min and max.

When you use these functions, make sure you refer to the library first:

```
math.sqrt(4) -> 2
```

There are hundreds of libraries you can use, all with their own use cases. You can find a list of Python libraries here.

### User-defined

As well as using Python’s existing functions, we can create our own functions to use whenever we need to within our program.

You can create a function using the def command:

```
def add_five_to_seven():
    return 5 + 7
```

Whenever we call this function, it will add 7 and 5 together and return the result back to us.

### Parameters

We can pass parameters to our functions to be used throughout:

```
def add_five(number):
    number_plue_five = number + 5
    return number_plus_five
```

This function expects a number to be passed in as a parameter when it is called. It will then take that number, add 5 to it and return the new number.

### Variables

If you define a variable inside a function, it is not accessible outside it.

# Files

Now we will look at how to read, write and edit text files in Python.

There are many different types of file that you're probably used to using every day. Pdf, python, mp4 and text files are just a few of these. Python can work with most types of files, but some of them require you to import extra modules.

File types are identifiable by their file extension, so python files will always end with .py and plain text files will end with .txt.

We will be working with plain text files (.txt), as they are simpler for beginners.

## Opening a File

Opening a file is simple. There are three things which you need to consider:

- file - the name of the variable in which we will store the file
- filename - the name of the file which we are working with (including the file extension)
- mode - the mode in which we want to open the file, which can be read-only (r), write-only (w), read and write (r+), or append (a)

```
file = open("filename", "mode")

file.close()
```

Once you've finished using a file, you need to close it.

Note: using the syntax above for closing files will always close the most recently opened files.

## Reading

There are three main ways to read files:

- readline() - reads the current line and then moves on to the next line
- read() - reads from the current line to the end of the file
- readlines() - reads the whole file

```
file = open("filename.txt", "r")

lines = file.readlines()

print(lines)

file.close()
```

You can store the data from a file in a variable in Python, to do anything you want with it. In this example, we are printing that data to the screen.

When you read more than one line, it is stored in a list, so you can iterate through that list to use each individual line.

## With

We can use a with statement to open the file and not have to worry about closing it.

```
with open("students.txt") as file:
        lines = file.readlines()


for line in lines:
   print(line)
```

We can still use the data in the file by storing it in a variable.

## Writing / Appending

We use the write function to add text to a file, but remember that, when in write mode, you will delete everything currently in the file.

```
with open("filename.txt", "w") as file:
for n in range(1,11):
                newline="This is a new line \n"
                file.write(newline)
```

This will write “This is a new line” 11 times. If we were to change the ‘w’ mode to ‘a’, the program would then add 11 new lines to the file at the end.

## Plotting Values on a Graph

You can use a Python package called matplot to draw graphs of given values. To use this, you need to import the package and the specific function you want.

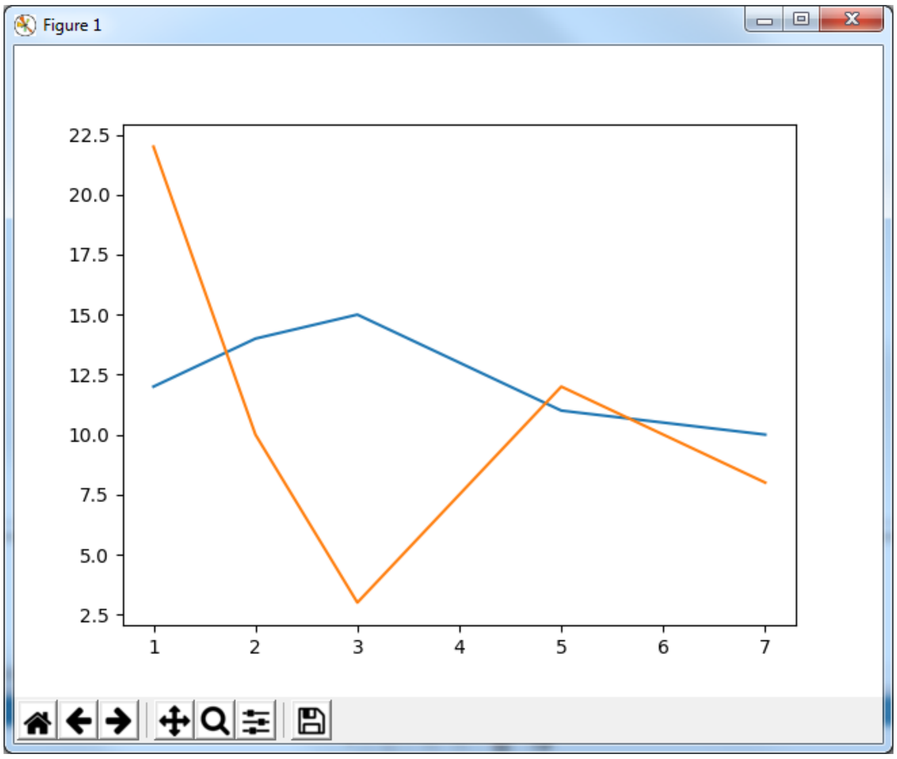
```
import matplotlib.pyplot as plt
```

You can then use the plt.plot() function, which takes two lists and will plot the graph based on the integers in the lists, the first list being the x values and the second list being the y values.

```
plt.plot([1,2,3,5,7],[12,14,15,11,10])
plt.plot([1,2,3,5,7],[22,10,3,12,8])
```

You can then show the graph using the show function:

```
plt.show()
```



## Pyodbc

Pyodbc is a Python module that helps you connect to databases. You can install it either using pip or through Visual Studio.

```
import pyodbc
```

We can establish a connection with the database using a connection string:

```
connectionString = r'DRIVER={ODBC Driver 13 for SQL Server};SERVER=.\SQLExpress;DATABASE=qastore;Trusted_Connection=yes'

conn = pyodbc.connect(connectionString)

conn.close()
```

This connection string defines the Driver we want to use to create the connection, the server we want to connect to, and which database to work with.

Notice that we close the connection once we have finished. In between the connect and close commands is where we would interact with the database.

### Read

To read from a database, we need an SQL command, which we can store in a variable as a string:

```
sqlStr = "SELECT * FROM customers WHERE city='London'"
```

We then need a cursor to execute the commands:

```
cur = conn.cursor()

result = cur.execute(sqlStr).fetchall()
```

We now have all the customers that live in London in the result variable. Try printing them as follows:

```
for row in result:
    print(row)
```

It is better practice to use functions for this task, as it's likely you'll need to query your database several times.

```
def readData(sql):
    conn = pyodbc.connect(connectionString)
    cur = conn.cursor()
    result = cur.execute(sql).fetchall()
    conn.close()
    return result
res =  readData("SELECT * FROM company WHERE county='Devon' ")
```

Now we can make many queries to our database and change the query to any SQL command.

## Updating a Database

The process of updating a database is very similar to reading a database's data, but there are some minor changes of which you should be aware.

Let's say our SQL command is:

```
sqlStr ="""INSERT INTO company
     (company_no, company_name, tel, county, post_code)
VALUES (5000,'QA','0207 888555','Devon','SE8 5ER')"""
```

We connect to the database and define the cursor exactly the same way:

```
conn = pyodbc.connect(connectionString)
cur = conn.cursor()
```

But when we execute the command, we don't need fetchall() on the end as there aren't any records to retrieve:

```
cur.execute(sqlStr)
```

We also need to save the changes we have made with the commit function:

```
conn.commit()
conn.close()
```

## Exception Handling

Sometimes in Python we can get errors (or exceptions). If we get an error in our programs but we want the program to continue to run and not fail, we need to handle the exception.

We handle exceptions with the try and except blocks:

```
try:
        print(x)
except:
        print(“X isn’t there”)
```

Try the code above, if we tried to print the variable x without defining it first, you will get an error. However, if you try and except the error, you’ll get your message and the program will run without errors.

## Classes

Object Oriented Programming (OOP) has four principles (which we'll discuss in a later module) that allow us to define complex types, known as classes.

From these classes, we create objects on which we can operate.

class - a set of attributes (data) and methods (code) which we can use to create objects

object - a usable instance of a class

Since Python is an object-oriented language, we can make use of these.

A simple example of a class is:

Class: - Dog

Attributes: - Breed

- Weight
- Energy

From this class, we can create usable objects:

Object: - Bilbo Waggins

Breed: Labrador

Weight: 80lbs

Energy: Low

### Classes in Python

This is what a class looks like in Python:

```
class Dog:

  energy = "high"
  def speak(self):
    print("woof")

bilbo_waggins = Dog()
print(bilbo_waggins.energy)
bilbo_waggins.speak()
```

The above example defines the class Dog, sets the attribute energy equal to "high", and defines the method speak to print "woof".

Then sets the object bilbo_waggins to be a Dog.

Printing bilbo_waggins' energy will print "high", and calling the method speak() will print "woof".

We can redefine the attribute if our Dog doesn't have high energy.

```
chewbarka = Dog()
chewbarka.energy = "low"
print(chewbarka.energy)
chewbarka.speak()
```

### Class Constructor

The class constructor allows us to set attributes when we've created an object.

Notice that when we defined the class Dog, we did not use parenthesis. However, when we defined the objects, chewbarka = Dog(), we did.

The method that we are calling here is the constructor, which is the method which is responsible for creating the instance of the class.

In our example, it doesn't do a lot, but it can!

```
class Dog:

  def __init__(self, name, breed, energy):
      self.name = name
      self.breed = breed
      self.energy = energy
```

Every time we construct a new object of this class, we must pass in name, breed and energy parameters as a reference to the current instance of the class.

dog1 = Dog("Ross Barkley", "Jack Russel", "High")

### Self

The self parameter is a reference to the current instance of the class. Class data members are usually set as default and belong to the class, whereas instance data members are unique to the instance of the class. We need a way to map one to the other, and that's what the self parameter does.

Note that the parameter doesn't need to be called self, that's just best practice, but it must be the first parameter of any class function!

## Class Methods

There are some useful methods to know:

- getattr(object, name) - Access the named attribute of a specific object
- hasattr(object, name) - Check if a specific object has a named attribute
- setattr(object, name, value) - Set the named attribute of a specific object as the value specified
- delattr(object, name) - Delete the named attribute of a specific object

## Leading and Trailing Underscores

### Single Leading Underscore

Used to signify a private or internal variable, you shouldn't access these variables. For example, _money = 1,000,000 as someone's balance is private.

### Single Trailing Underscore

We use this to avoid conflicts with Python keywords. For example, we might define class_ = "QAC Intake", but class is a keyword in Python so we need to avoid conflicting with it.

### Double Leading Underscore

A double leading underscore applies name mangling to the class. This is where the interpreter changes the name of the variable to make it harder to create collisions when the class is extended.

We access variables that have double leading underscores in the following way:

```
_ClassName__VariableName.
class Feline:
    __family = "Felidae"


kitty = Feline()
print(kitty._Feline__family)
```

This would print "Felidae", the value of the __family variable.

Whereas print(kitty.__family) wouldn't print anything.

# Object Oriented Programming (OOP)

There are four principles to adhere to when practicing OOP:

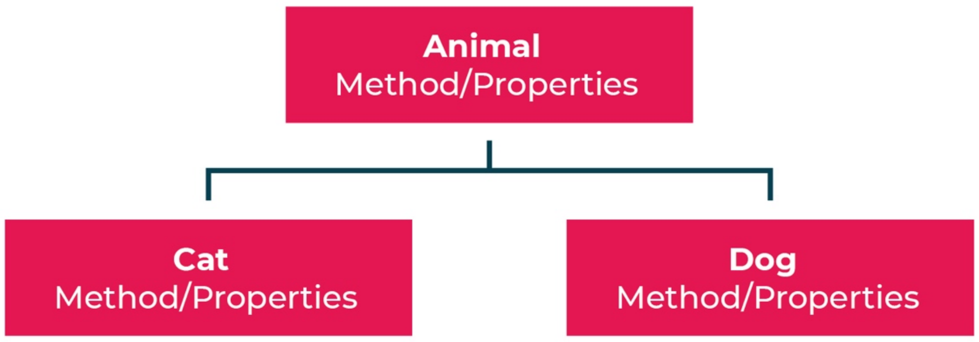
- Inheritance - One object acquires all the properties and behaviours of a parent object
- Polymorphism - Performing a single action in different ways
- Abstraction - Hiding the implementation details and showing only functionality to the user
- Encapsulation - Wrapping code together and data together into a single unit for data integrity

## Inheritance

When working in an OOP style, we can inherit attributes and methods. The superclass is the class being inherited from. The subclass is the class that is inheriting attributes and methods.

Let's take a look at this in a real-world modelling example: We inherit traits from our family such as eye colour, hair colour, etc. That is why superclasses and subclasses are commonly referred to as parent and child classes.

So why do we use inheritance? Imagine that we have a Cat and a Dog class. Both these classes need to represent the properties and methods that belong to an animal. This means that the Dog and Cat both need to have all this code:



Let's take a look at another example, but this time we'll show you how to use inheritance:

```
class Athlete:
  def __init__(self, name):
    self.name = name
```

```
class Footballer(Athlete):
    def __init__(self, name, team):
        super(Footballer, self).__init__(name)
        self.team = team

Player1 = Footballer("Jay", "Droylsden FC")
```

Notice when we create the Footballer class, we pass the Athlete class as a parameter.

## Polymorphism

Polymorphism means 'many forms'. The reason we use inheritance is so that a subclass can use the data members from a parent class. However, different child classes may act differently in regards to a data member inherited from the superclass. This is polymorphism.

So if we consider our Animal superclass, with cat and dog subclasses, we can have a method called speak. A cat's response to the method would be meow and the dog's class response would be woof.

There are two types of polymorphism: overloading and overriding.

## Overloading

Overloading is the action of defining multiple methods with the same name:

```
class Bird:
    def __init__(self, wingspan):
        self.wingspan = wingspan

Eagle = Bird(104)

len(Eagle)
```

This will produce an error, because the len method has no logic assigned to it in the Bird class.

```
class Bird:
    def __init__(self, wingspin):
        self.wingspan = wingspan
    def __len__(self):
        return self.wingspan

eagle = Bird(104)

len(eagle)
```

This time, the logic of len returns different results depending on the object passed into it.

## Overriding

Overriding is the process of overriding a behaviour from a parent class:

```
class Animal:
    babies = 0

    def reproduce(self):
        self.babies += 1

class dog(Animal):
    def reproduce(self):
        self.babies += 6

john = dog()
john.reproduce()
print(john.babies)
```

In this piece of code, the child class of dog overrides the reproduce method of the parent class Animal.

## Encapsulation

Encapsulation refers to bundling together data with methods that will operate on that data.

Usually, we keep variables private so that there isn't any direct access to them; they have to be retrieved or manipulated through public methods.

```
class BankAccount:
  _money = 0

  def deposit(self, money, password):
    if (authenticate(password)):
      self.money += money

  def withdraw(self, money, password):
    if (authenticate(password)):
      self.money -= money
```

In the above example, no one has direct access to the variable _money. The only way to change that variable is through the methods deposit and withdraw. For these methods to change the _money variable, the user must have the correct password.

### Abstraction

Abstraction allows us to create the lowest level blueprint of a class without anyone being able to create an instance of that class.

For example, animals do not exist solely as just an animal. They have other defining attributes, such as their biological class (mammal, reptile, etc.) or eating habits.

Realistically, we want each subclass of Animal to apply their own way of eating.

In Python, the in-built Abstract Base Classes (abc) library contains the infrastructure for us to create our own abstract classes. Any abstract class we create is a subclass of the abstract base class, ABC:

```
from abc import ABC, abstractmethod


class Animal(ABC):
  @abstractmethod
  def eat(self):
    pass


class Mammal(Animal):
  def eat(self):
    print("nom nom")
```

We use the @abstractmethod decorator so that any subclass created must have its own implementation of that method. In the example above, the eat method is an abstract method, so it must be defined for the Mammal class.

## Linting

Linting points out syntactical and stylistic problems with your Python code. This can help you spot errors before they happen, as well as make sure you're following best coding practices.

The linting tool we are going to be using is called Pylint, which you can install with pip as follows:

```
pip3 install pylint
```

Once you have pylint installed, you can write a very simple program to test it out:

```python
from math import sqrt


def square_times_two(x):
  return sqrt(x) * 2
```

Save this code in a file called numbers.py. In your terminal, in the same directory as your file, run this command:

```
pylint numbers.py
```

This should be returned:

```
************* Module numbers
numbers.py:1:0: C0114: Missing module docstring (missing-module-docstring)
numbers.py:4:0: C0103: Argument name "x" doesn't conform to snake_case naming style (invalid-name)
numbers.py:4:0: C0116: Missing function or method docstring (missing-function-docstring)

-----------------------------------------------------------------
Your code has been rated at 0.00/10
```

This is pylint giving your code a score out of 10 and telling you how to improve the score.

In this example, the first thing we can fix is our variable name:

```
from math import sqrt
def square_times_two(number):
  return sqrt(number) * 2
```

From here, we will get this:

```
************* Module numbers
numbers.py:1:0: C0114: Missing module docstring (missing-module-docstring)
numbers.py:4:0: C0116: Missing function or method docstring (missing-function-docstring)


------------------------------------------------------------------
Your code has been rated at 3.33/10 (previous run: 0.00/10, +3.33)
```

As you can see, our score has gone up. Notice the next suggestions are about docstrings (comments). This means that we have two options: we can change our code to follow best practices with comments, or we can ignore the advice. Let's take a look at how to do both.

To fix the code, we need two comments: one with the function and one with the import:

```
""" import sqrt from math library """
from math import sqrt
def square_times_two(number):
  """
  This function finds the sqrt of a number and multiplies it by 2

  Parameters: number (int)

  Returns: (int) The number after its been squared rooted and times by 2
  """
  return sqrt(number) * 2
```

Now pylint will say:

```
-------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 3.33/10, +6.67)
```

Our second option is to ignore the comments, which we can do by running this command:

```
pylint numbers.py --disable=C,R
```

# Testing

According to the ANSI/IEEE 1059 standard, testing is 'a process of analysing a software item to detect the differences between existing and required conditions (i.e. defects) and to evaluate the features of the software item'.

In other words, testing is the activity of running code under various scenarios and circumstances to highlight if and where defects exist. But you don't always have to test to find defects - you can also use testing to compare the application's characteristics to a set of business requirements and rules.

## Why We Test

Imagine we are copying a file from one location to another - this is a simple task that people do everyday. But what could go wrong when you're doing it?

- File could not be found
- Destination unavailable
- File already in use
- Incorrect permissions
- Network fail during transmission
- File corrupted during transmission
- Disk space availability
- Blocked by firewall

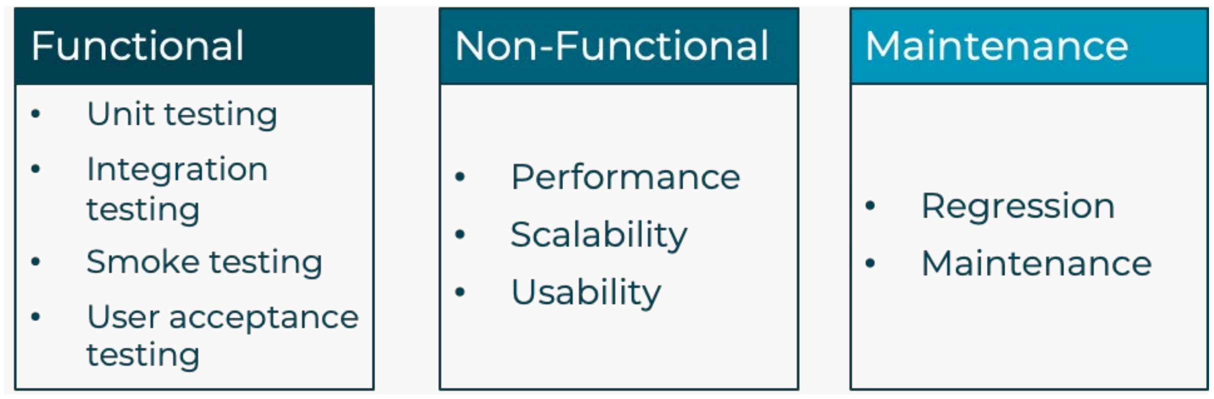
So if we can come up with all these possible errors for one simple task, imagine all the possible errors that could occur for 200 lines of code!

There are three main reasons that we want to fix these errors, rather than releasing the code as is:

1. The cost of fixing defects when a product is in production (live) is far greater than the cost of fixing defects identified in an earlier phase.
2. Releasing buggy code can often have an impact on customer faith and business reputation.
3. Testing aids in providing quality assurance to a business.

## Types of Testing

There are three different types of testing, and many methods to carry out the different types:



| Functional | Non-Functional | Maintenance |
|---|---|---|
| • Unit testing<br>• Integration testing<br>• Smoke testing<br>• User acceptance testing | • Performance<br>• Scalability<br>• Usability | • Regression<br>• Maintenance |

- Functional - If I click a button, does it perform the correct action?
- Non-functional - If 10,000 people click the same button, is the response time acceptable?
- Maintenance - Once changes have been made, do all previous tests still pass?

## Unit Testing in Python

### Automated Testing

There are many reasons why we should automate tests, but the main benefit is repeatability.

This means that we can run tests whenever a change is made and not have to completely rewrite the tests from scratch each time. We can also be confident that if the changes pass the tests, the software will work with minimal defects.

The easiest way to achieve this is through unit tests. Unit tests are scripted tests in Python to verify the unit of functionality. Python accommodates for this through a number of tools, but we'll be focusing on pytest.

### Pytest

Pytest is a framework that allows for automated testing in Python. You can install it through pip.

### Asserting

Assert statements allow us to establish whether a test has passed or failed by asserting certain states of the code.

### Writing Tests

When you run the pytest command, it will look for filenames in the form of test_[name].py or [name]_test.py.

In these files, your tests must be in the form of functions and, similarly to the filename, the name of the function must be in the form def [name]_test(): or def test_[name]():.

In these files, you will need to import the functions you wish to test, so the testing file and the original python file must be in the same directory.

The command to run the test is pytest.

## Separating Tests and Code

We currently have both our tests and code in the same python file, but this isn't easy to maintain.

We want to keep the functions we wish to test separate from the functions that test them. We can do this by using files or directories.

### Files

The first way to separate our functions is into different python files (.py).

For example, if we have two python files, double.py and test_double.py, we will first have to import the double file to the test_double.py.

Then to use the function, we have to reference the file followed by the function.

test_double.py:

```
import double

def test_answer():

    assert double.func(6) == 10
```

double.py:

```
def func(num):

    return num * 2
```

### Directories

If we have more than one function file or more than one test file, we may want to separate our functions and tests even more by having them in separate directories.

We can create two new directories: one called 'programs' to store the functions files, and one called 'tests' to store the tests files.

In order to import the functions as before, we need to make sure that there is an empty __init__.py in both the programs and tests directories. This __init__.py file links the two directories and allows us to use the following command in our test files.

test_double.py:

```
from programs import double

def test_answer():

    assert double.func(6) == 10
```

double.py:

```
def func(num):

    return num * 2
```

## Test Driven Development (TDD)

TDD is a development method that programmers often use where we first write test cases for the code we want to develop, then we write the code to pass the test.

TDD allows us to write much cleaner, simpler code that is bug free. It forces developers to write the code to pass the test, as opposed to writing the code and then writing a test that passes but might not actually test the functionality of the code.

In this more traditional style of development, testing is done as an afterthought, which causes the code to have more bugs and makes it more difficult to maintain. Traditional development can also cause developers to skip testing entirely as it is thought of as being the testers' job. This can cause the introduction of technical debt, which is the estimated cost, both in terms of money and time, of fixing the bugs within the code.

Working within TDD guidelines allows us to implement small amounts of code at a time easier, making sure that we reduce code duplication and keep it simple.

## The Steps of TDD

The TDD steps are known as the 'red-green-refactor' workflow:

1. Add a new test.
2. Run the test suite to see if the test passes based on current code, or if it fails and thus needs new code.
3. Write the code needed to fulfil the requirement so that the test passes. In this step, we want to reduce the amount of code duplicated, make it as simple as possible, and not write more code than is needed.
4. Run the test again to see if our new code works as expected. If it does, go back to Step 1 and add the next test; if it doesn't, repeat Step 3.
5. Refactor the code to remove any duplicated code, make cleaner and more maintainable, and simplify it. Refactoring is a technique used to restructure an existing body of code by altering the internal structure without changing its external behaviour.
6. Repeat Steps 1 to 5 for the next piece of functionality.

## The SOLID Principles

American Software Engineer Robert ‘Uncle Bob’ Martin outlined five principles that developers should follow, in order to keep our work understandable, flexible and maintainable. You can remember these principles using the mnemonic ‘SOLID’.

### S - Single Responsibility

When talking about **responsibilities** in programming, we essentially talk about **reasons for something to change**.

If a class has a single responsibility, then it only has one reason to change.

The single responsibility principle states that a class should only ever have one purpose, and therefore only one reason for it to ever need to change.

### O - Open-Closed

When talking about the way in which our classes are laid out, we generally say that they should be **open for extension** but **closed for modification**.

We want developers to extend and add to your functionality, without having to directly modify our classes - this is usually because our previous code has gone through testing, and we can be sure that it works!

### L - Liskov Substitution

The Liskov substitution principle states that functions that use pointers to **base classes** (parent classes) must be able to use objects of **derived classes** (child classes) without knowing it.

### I - Interface Segregation

The interface segregation principle states that you shouldn’t be forced to depend on interfaces containing methods that you aren't going to use.

Essentially, you should only ever have methods inside an interface together if they all need to be implemented at once.

### D - Dependency Inversion

The dependency inversion principle states two key points:

- High-level modules shouldn’t depend on low-level modules - both should depend on abstractions, such as interfaces
- Abstractions shouldn’t depend on details (concrete implementations) - instead, details should depend on abstractions

Essentially, both higher- and lower-level modules should depend on the same abstractions, rather than on each other.