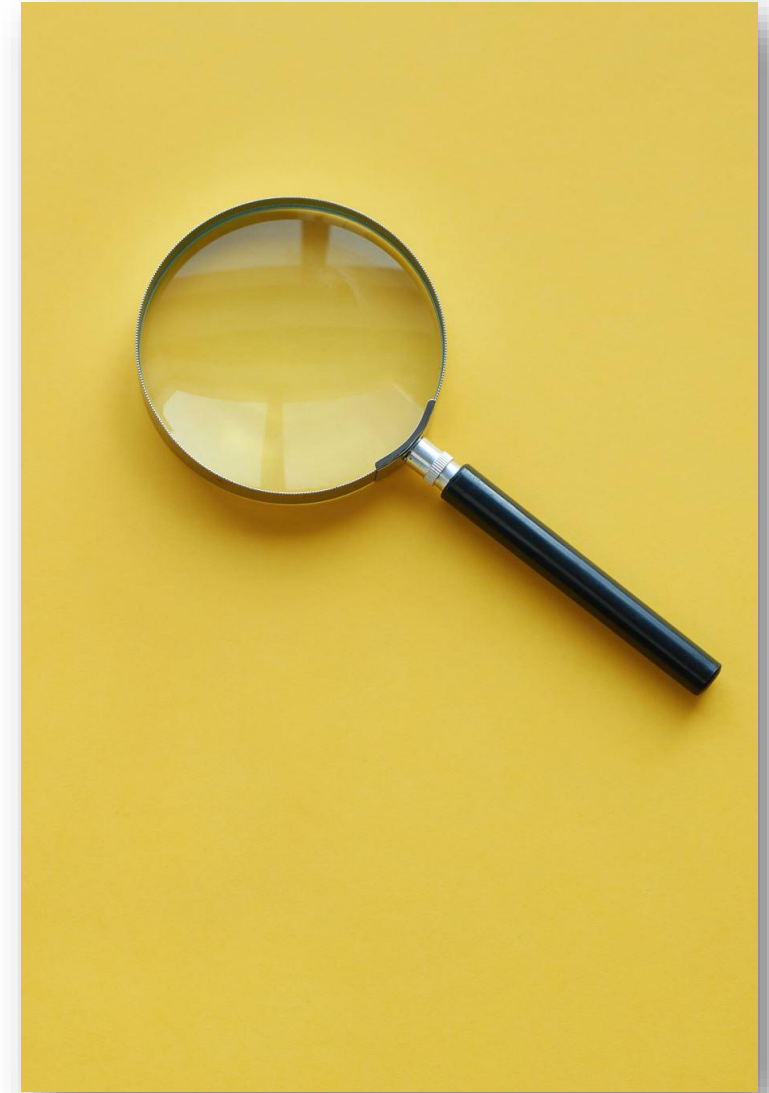


# Week 4 Session 1

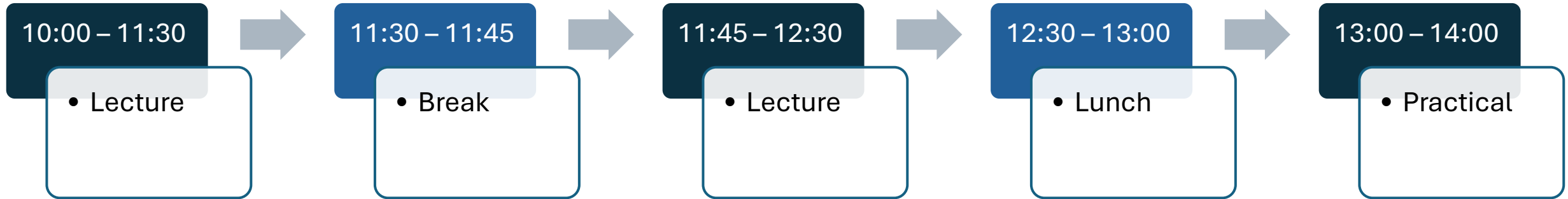
## Introduction to Python

# Topics

- An introduction to Python programming.
- Understand programming concepts.
- Understand data types, variables and data structures.
- Gain practical understanding of core programming tools.
- Implement Python programming through practical exercises.



# Timing



Keyword	Description
Sequential	Sequential code means that <b>it is accessed by a single thread</b> . This means that a single thread can only do code in a specific order, hence it being sequential. The other thing is concurrent code, multiple threads may access the same code synchronously.
Iteration	In programming specifically, iterative refers to <b>a sequence of instructions or code being repeated until a specific result is achieved</b> . Iterative development is sometimes called circular or evolutionary development.
Conditions	Conditional statements are <b>used through the various programming languages to instruct the computer on the decision to make when given some conditions</b> . These decisions are made if and only if the pre-stated conditions are either true or false , depending on the functions the programmer has in mind.
Data Structure	A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways.

Keyword	Description
Subroutines	A subroutine is a named section of code that can be called by writing the subroutine's name in a program statement. Subroutines can also be referred to as procedures or functions.
Logical Operator	Logical operators can be used to create complex conditions and control the flow of your program based on multiple conditions. They are commonly used in conditional statements such as if and while.
Math Operator	In Python, mathematical operators are used to perform mathematical operations on numbers. The basic mathematical operators in Python are + for addition, - for subtraction, * for multiplication, / for division.



# Python

## General principles

# History of Python

- Designed by Guido van Rossum and released in the early 1990's
- Based off the “ABC” programming language
- The name was inspired by the BBC comedy show Monty Python's Flying Circus
- Currently on Python version 3
- One of the most popular programming languages
  - Beginner to advanced
- Wide range of applications
  - Web Development
  - Data Analysis
  - Data Science and Machine Learning





# What is Python?

- Python is a popular, high-level, interpreted programming language
- High-level
  - Abstracts away low-level details
  - Easier to read and write
- Interpreted programming language
  - Executed line by line
- Simple, easy to learn syntax
  - Emphasises readability
  - Reduces the cost of program maintenance
- Freely distributed
- Can be used on multiple platforms





# What Is Python?

- Object-oriented
  - Objects and classes help organise code
- Supports modules and packages
  - Program modularity
  - Code reuse
- Extensive standard library
- Built-in data structures
  - Lists, dictionaries, etc.
- Scripting or glue language
  - Connect existing components together



# Important notes

- Python is case sensitive.
  - Code will not work without correct case.
- “Text” values need to be in “” double quotations, numbers do not.
- Variable names cannot be any Python reserved keywords e.g. print, input, etc.
- Files cannot be saved using Python reserved keywords.
- Comments can be used to explain code.
  - Start the line with #.
  - Lines starting with # will not be run.
  - Provides clarification of a function or variable to the human coder.

```
▶ ▼  
# Asks the user to enter their name  
name = input("Please enter you name: ")  
print("Hello", name)
```



# What Is Python Used For?

- Web Development - Frameworks such as Django and Flask
- Data Analysis - NumPY and Pandas
- Data Visualisation - Matplotlib and Seaborn
- Machine Learning - Scikit-Learn, NLTK, Tensorflow, Pytorch
- Web Scraping - BeautifulSoup
- Computer Vision - OpenCV Image Library
- Internet of Things (IoT) - Raspberry Pi + Python
- Game Development - PyGame

```
Terminal

$ pip install scrapy
$ cat > myspider.py <<EOF
import scrapy

class BlogSpider(scrapy.Spider):
    name = 'blogspider'
    start_urls = ['https://www.zyte.com/blog/']

    def parse(self, response):
        for title in response.css('.oxy-post-title'):
            yield {'title': title.css('::text').get()}

        for next_page in response.css('a.next'):
            yield response.follow(next_page, self.parse)
EOF
$ scrapy runspider myspider.py
```

Example Image, Web scraping: <https://scrapy.org/>

# Python

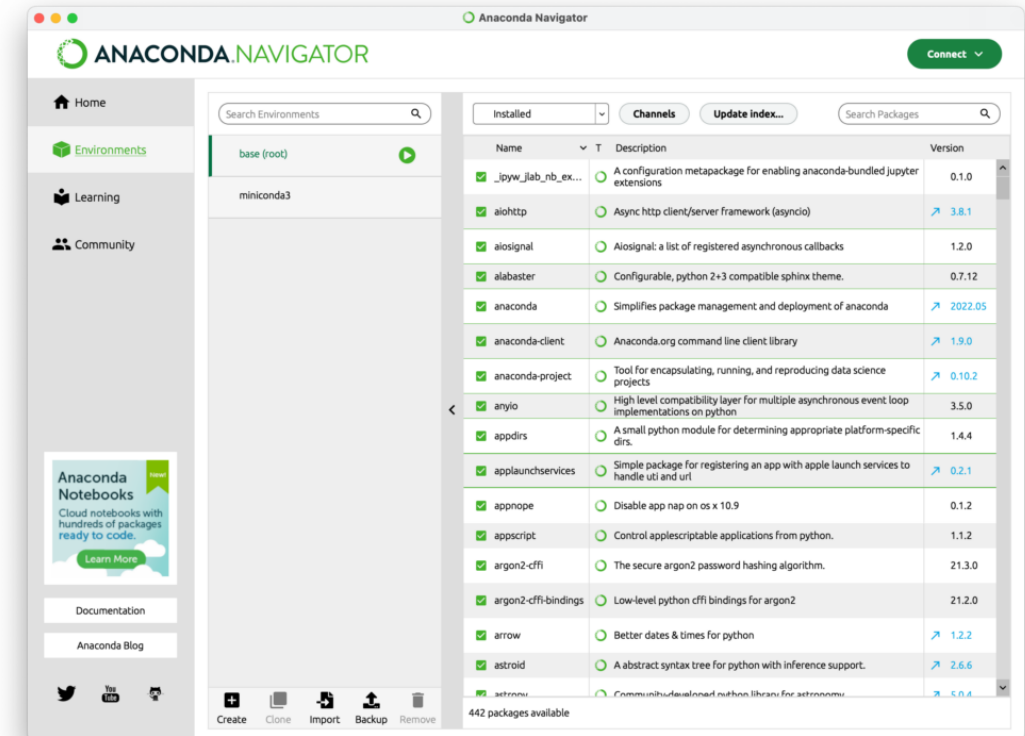
## Tools



# Anaconda

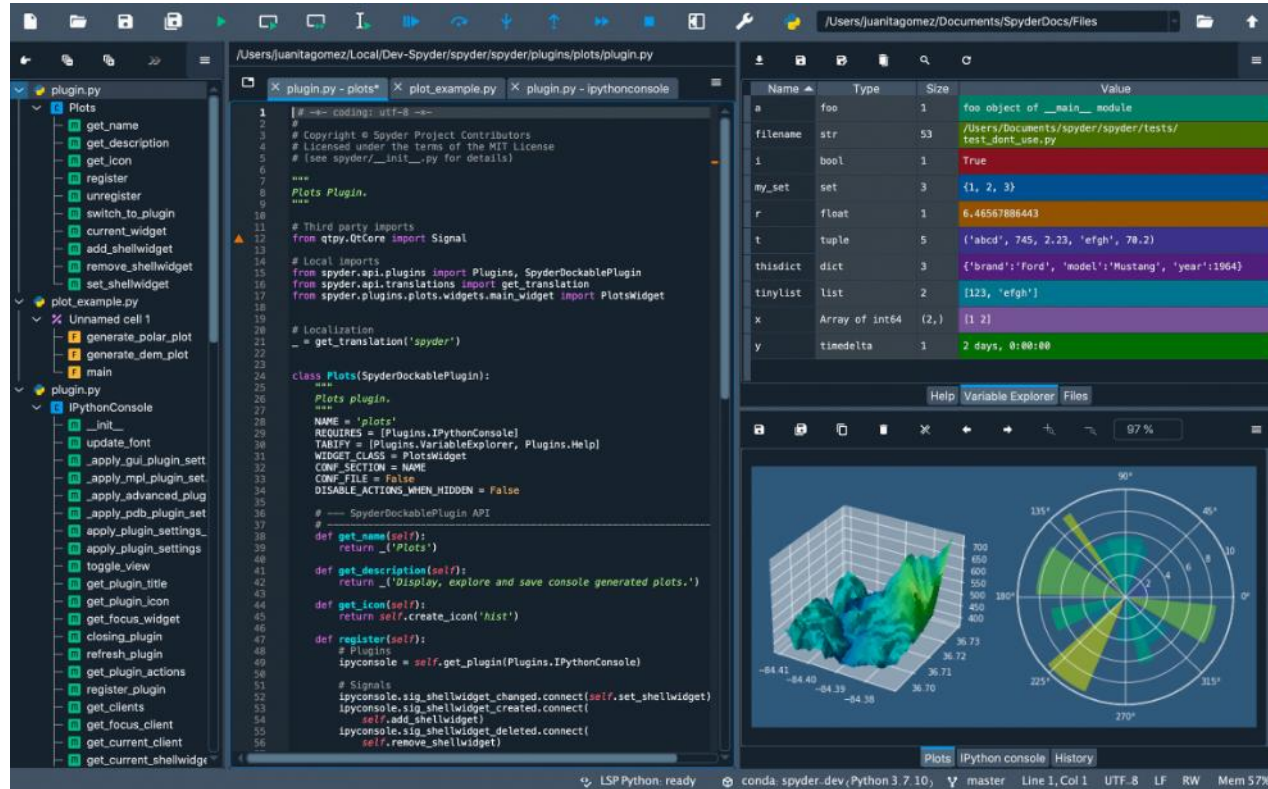


- World's most popular Python data science platform
  - Open source
  - Everything you need 'out of the box'
  - <https://www.anaconda.com/download/>
- Graphical modules and packages manager
  - Ease of use
  - Abstracted from command line
  - Discover previous unknown packages
  - Create and manage multiple environments
- Anaconda includes
  - Spyder – Scientific Python Development Environment
  - Jupyter



# Spyder

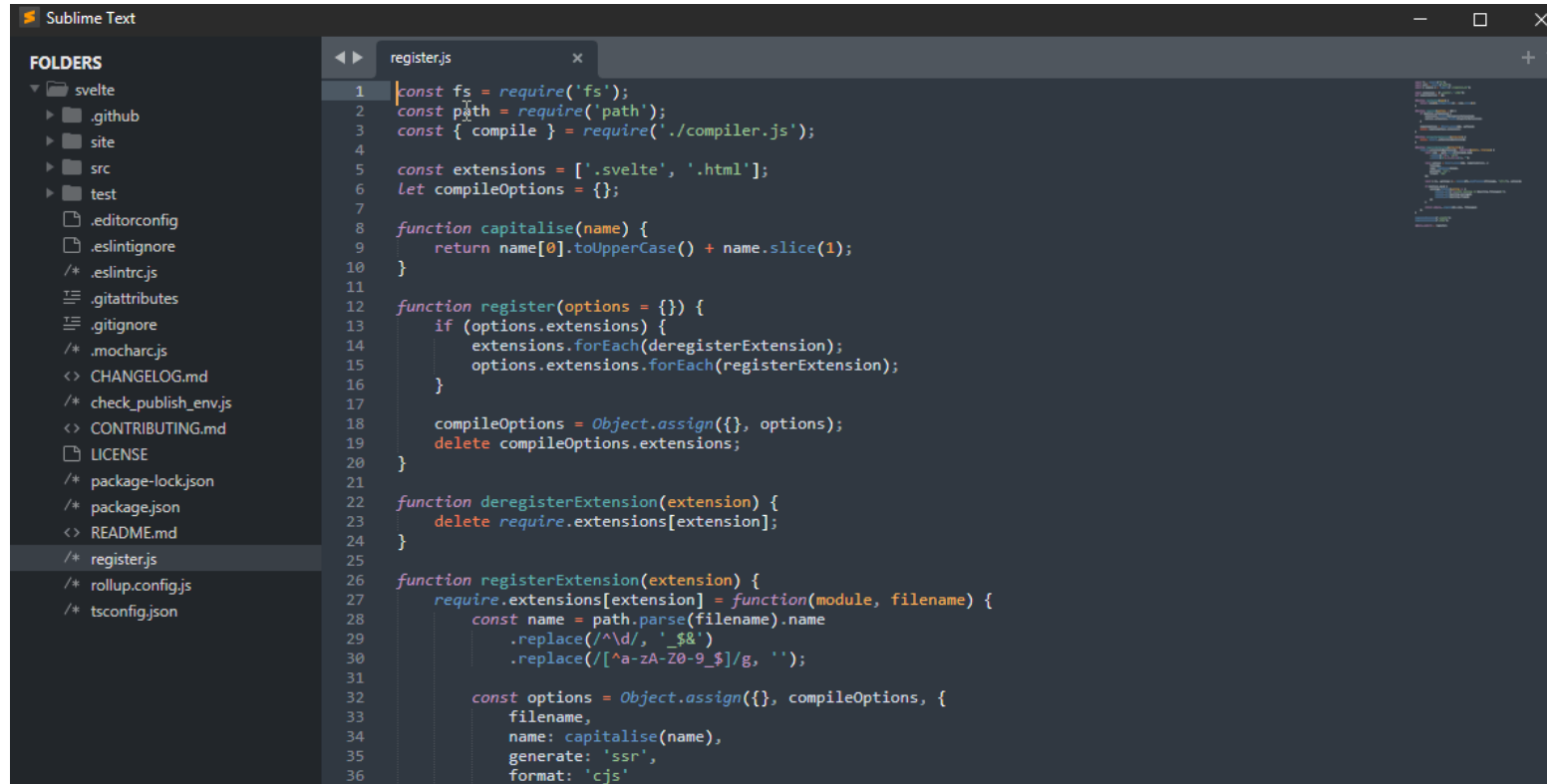
- Spyder is an Integrated Development Environment (IDE)



Spyder IDE: <https://www.spyder-ide.org/>

# Sublime

- Sublime is a text editor - Free trial before requiring purchase



The screenshot shows the Sublime Text editor with a dark theme. On the left, a sidebar displays a file explorer with a tree view of folders and files. The main editor area shows a file named 'register.js' with the following JavaScript code:

```
1 const fs = require('fs');
2 const path = require('path');
3 const { compile } = require('./compiler.js');
4
5 const extensions = ['.svelte', '.html'];
6 let compileOptions = {};
7
8 function capitalise(name) {
9   return name[0].toUpperCase() + name.slice(1);
10 }
11
12 function register(options = {}) {
13   if (options.extensions) {
14     extensions.forEach(deregisterExtension);
15     options.extensions.forEach(registerExtension);
16   }
17
18   compileOptions = Object.assign({}, options);
19   delete compileOptions.extensions;
20 }
21
22 function deregisterExtension(extension) {
23   delete require.extensions[extension];
24 }
25
26 function registerExtension(extension) {
27   require.extensions[extension] = function(module, filename) {
28     const name = path.parse(filename).name
29       .replace(/\d/, '_$&')
30       .replace(/^[^a-zA-Z0-9_$]/g, '');
31
32     const options = Object.assign({}, compileOptions, {
33       filename,
34       name: capitalise(name),
35       generate: 'ssr',
36       format: 'cjs'
```

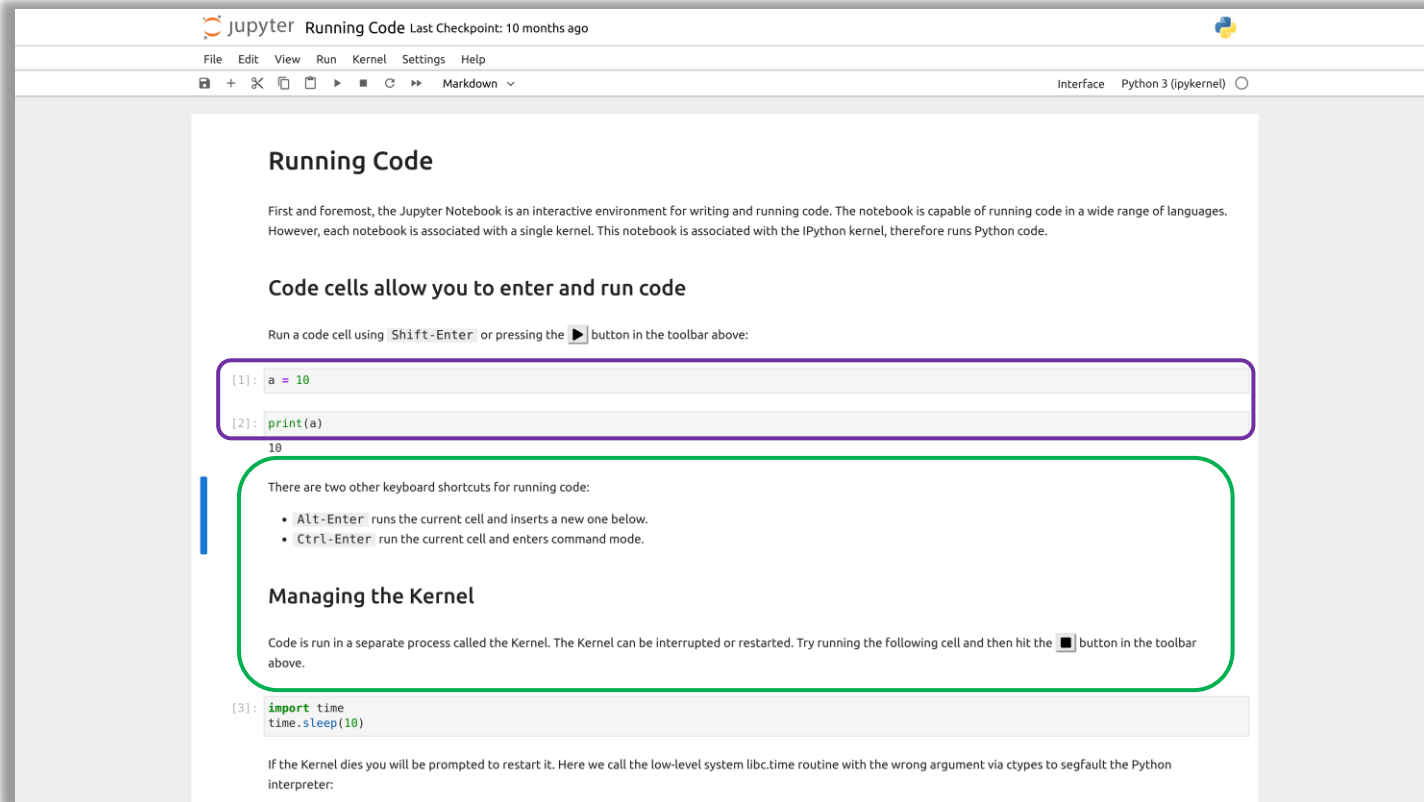
Sublime Text Editor: <https://www.sublimetext.com/>



# Jupyter Notebook



- A shareable document.
- A fast interactive environment.
  - Computer code.
  - Markup language.
  - Data.
  - Visualisations.
- Kernel.
  - Remembers run cells state.
  - Can run small chunks of code.
  - No need to re-run whole codebase.
- Avoids having to reload datasets each run of the code.
- Explore and visualise data quickly.
- Add explanatory comments.



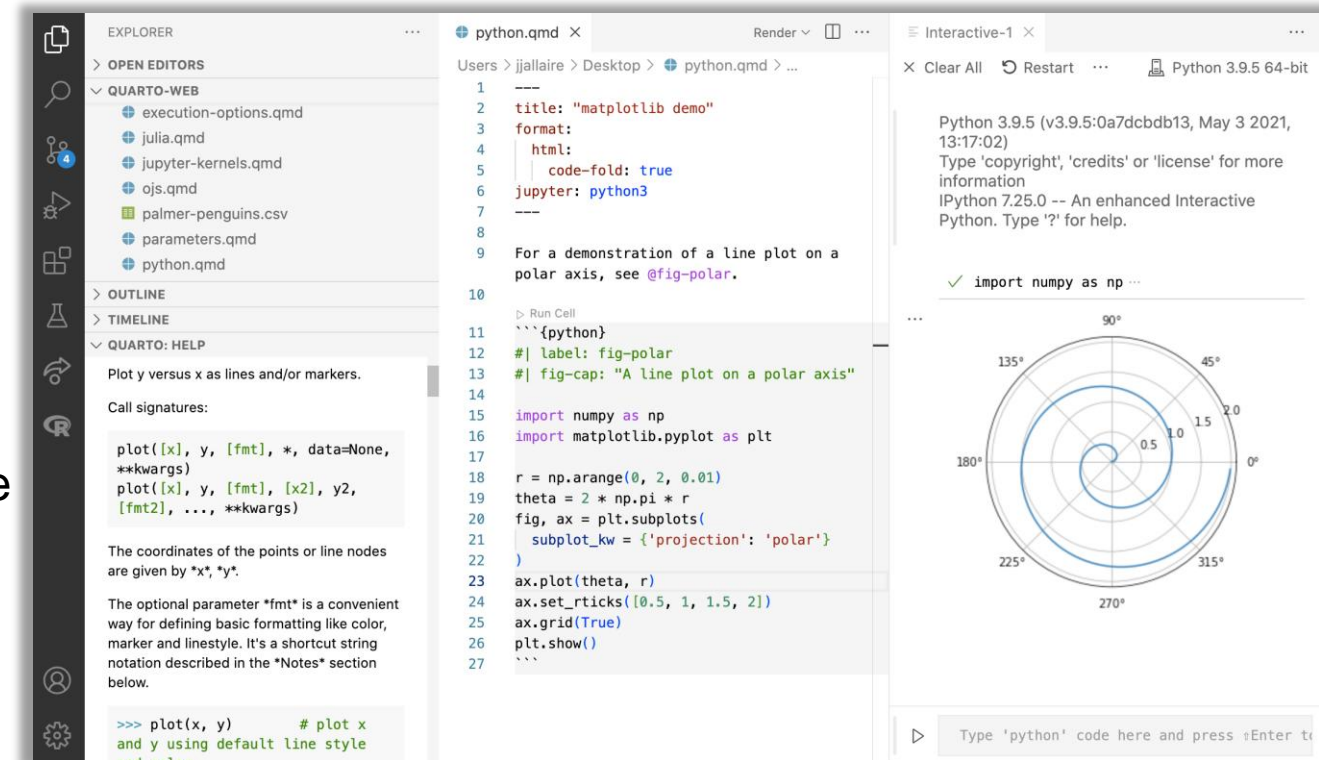
Jupyter Notebook: <https://jupyter.org/>  
Documentation: <https://docs.jupyter.org/>

# Visual Studio Code



Visual Studio Code

- Powerful data analysis and data science IDE
- Quickly set up an environment and start analysing your data
- Native support for Jupyter notebooks
- Easily create, open, and save Jupyter notebooks
- Work with Jupyter code cells
- View, inspect, and filter variables using Variable Explorer and Data Viewer
- IntelliSense support
  - Autocomplete suggestions
  - Libraries, variables, etc
- A lighter weight tool than Visual Studio



VSCode: <https://code.visualstudio.com/>

Download: <https://code.visualstudio.com/#alt-downloads>

# Python

Set up your preferred tools now



# Python PEP8 Styling

PEP8 is the official style guide for Python code.

Documentation: <https://peps.python.org/pep-0008/>

- Employers will often require the use of a styling code.
- Best practice for readable, consistent code.

## Indentation

- Avoid using tabs – not all platforms will read tabs correctly. Use 4 spaces per indentation level (will be read by all platforms correctly).

## Line length

- Limit all lines to a maximum of 79 characters, so everything can be seen in the same size panel.

## Blank lines

- Surround top-level function and class definitions with two blank lines.

## Naming conventions

- Use different naming styles for different types of objects;
  - Function and variables use 'Python case' *lower\_case\_with\_underscores*
  - Classes use *CapitalisedWords*

## Whitespace

- Use whitespace in expressions and statements to improve readability.
- Don't add too much whitespace if it's not helping to make your code more readable!



# Python

## Programming concepts



# Programming Concepts

- Programs consist of the same basic 'building blocks' assembled in different ways to achieve a particular goal
- Basic building blocks: variables, data types, sequences, selection and iteration
- Comparing and manipulation and decision making
- Logical, relation, arithmetic operators and error checking
- Randomness can be simulated e.g. selecting a random number or item from a list
- Precise instructions. **Computers follow code exactly**

## For humans:

A LEGO model is a good example of something that includes precise instructions.

- Must be followed to the letter
- Avoids disassembly for future parts
- Clearly readable for human users
- Usually human error if not assembled correctly
- Allows for different interpretations

## For computers:

- Follows instructions exactly as written
- No misinterpretations
- Cannot guess what instructions mean
- Unexpected behaviour is from human error in the coding of a program

# Syntax

- Each programming language has its own syntax rules.
- Syntax comprises of Python reserved keywords, characters and symbols.
- It is important to remember that **ALL** syntax needs to be correct for the program to run.
- PEP8 Styling aids in detecting syntax errors quickly. Example: We cannot use a python reserved keyword as a variable name

```
▶ # Asks the user to enter their name
print = input("Please enter your name: ")
print("Hello", name)

[8] ✖ 2.5s

... -----
TypeError                                Traceback (most recent call last)
Cell In[8], line 3
      1 # Asks the user to enter their name
      2 print = input("Please enter your name: ")
----> 3 print("Hello", name)

TypeError: 'str' object is not callable
```

```
▶ name = input("Please enter your name: ")
print("Hello", name)

[2] ✔ 3.2s

... Hello Mike
```

PEP8 Styling Documentation: <https://peps.python.org/pep-0008/>





# Types of Errors

## Syntax

- Correct syntax is critical.
- Computers do not guess how to run a program.
- They follow instructions exactly as written. If they cannot, the program will halt.

```
▶ # Asks the user to enter their name
print = input("Please enter your name: ")
print("Hello", name)

[8] 2.5s

-----
TypeError                                Traceback (most recent call last)
Cell In[8], line 3
      1 # Asks the user to enter their name
      2 print = input("Please enter your name: ")
----> 3 print("Hello", name)

TypeError: 'str' object is not callable
```

## Logical

- Code might run without apparent error
- Execution might not be as expected, e.g. incorrect math.
- Much harder to debug!
- Both examples below run without errors, but only one gives the correct result.
- Simple example, but imagine a more complicated equation

```
Task: Add '3' and '1' together and output the result

sum = 3 - 1
print(sum)

[1] ✓ 0.0s
... 2

Task: Add '3' and '1' together and output the result

sum = 3 + 1
print(sum)

[2] ✓ 0.0s
... 4
```



More examples: <https://www.geeksforgeeks.org/errors-and-exceptions-in-python/>

# Python

## Data types



Data Type	Description	Example
String	Strings hold a list of characters, must use single or double quotations when entered	"iPhone 14"
Integer	A whole number without a decimal point can be positive or negative	0, 6, 2880, -76, 42
Float	A number which includes a decimal point can be positive or negative	5.2, -7.355
Boolean	A Boolean variable can only have one of two values.	TRUE or FALSE



Python has built in data types

- **Integer <int>**
  - Whole numbers - 1, 2, 3, etc
- **Floating-point numbers <float>**
  - Decimal numbers e.g. 3.14 or 2.7
- **Strings <str>**
  - Text or a sequence of characters e.g. "Hello world"
- **Booleans <bool>**
  - True or False
- **Lists**
  - A collection of values in a single variable
- **Dictionaries <dict>**
  - Stored key-value pairs in a single variable

```
# Integer
type(7)
✓ 0.0s
int

# Float
type(3.14)
✓ 0.1s
float

# String
type('Hello')
✓ 0.0s
str
```

```
# Boolean
type(True)
✓ 0.0s
bool

# List
x = [1, 2, 3]
type(x)
✓ 0.0s
list

# Dictionary
y = {"a": 1, "b": 2}
type(y)
✓ 0.0s
dict
```

There are more data types available: tuples, sets, objects, classes. We will get to those!



# Variables

- A container for a value.
- The value can change during the program execution.
- Can be named almost anything (but not a Python reserved keyword).
- Variable name:
  - Must be unique.
  - Must start with a letter, not a number.
  - Must have no spaces.
- Assignment operator =
  - A single equals sign is used to assign a variable.
- Variable type assignment is automatic.
- Other programming languages might require definition of type.

```
# A variable (variable = value)
id = 1009
name = "Mike"
linked = True
```



# Changing Variables

- The value can change during the program execution.
- Variables only hold one value at a time.
- New data placed into a variable will overwrite the last data.
- Example: Keeping track of a game score.
  - Score starts at zero
  - Player gains 1 score
    - Take the current score and add 1
      - $\text{score} + 1$
    - Store new score into the score variable
      - $\text{score} = \text{score} + 1$
  - Score increases by 1

score 0

`score = score + 1` — code  
score 1

```
# Starting score
score = 0

# Player earns a point
score = score + 1

# Print score
print(score)
```

✓ 0.0s

1

# Type Casting

- The process of converting one type of variable to another.
- Code might not return values in a type you need e.g. needs to be a number type to perform math logic.
- Two types of casting
  - Implicit - Python automatically converts between types
  - Explicit - the user converts the data types
- Constructor functions are premade functions already available to type cast.
  - `int()`
  - `float()`
  - `str()`

```
# int to float  
num = 5  
num = float(num)  
  
print(num)
```

✓ 0.0s

5.0

```
# float to int  
num = 6.0  
num = int(num)  
  
print(num)
```

✓ 0.0s

6

```
# str to int  
text = '34'  
num = int(text)  
  
print(num)
```

✓ 0.0s

34



# Python

## Operators



# Input/Output Operators (I/O)

- Input from the user
- E.g. store the input into a variable named 'textValue'.

```
textValue = input("Enter a text value: ")
```

- Input is treated as string type by default.
- Typecast the string to type int.
  - Wrap the input() function with the int() function.
  - Nesting functions, the same as we did in Excel.

```
numValue = int(input("Enter a number: "))
```

- Output to display

- Print() function shows a message on screen

```
print("This is a message")  
✓ 0.0s  
This is a message
```

- \n is used as a line break

```
print("This is \na message")  
✓ 0.0s  
This is  
a message
```

- Print can also utilise variable names

```
answer = 5  
  
print("The answer is: ", answer)  
✓ 0.0s  
The answer is: 5
```

# Math and Logical Operators

- Math operators

Operator	Description	Example
+	Add	<pre>x = 3 y = 4 z = x + y print(z) # 7</pre>
-	Subtract	<pre>x = 3 y = 4 z = x - y print(z) # -1</pre>
/	Divide	<pre>x = 3 y = 4 z = x / y print(z) # 0.75</pre>
*	Multiply	<pre>x = 3 y = 4 z = x * y print(z) # 12</pre>

- Logical operators - evaluate multiple relational expressions to return a single value

Operator	Description	Example
and	All result True	<pre>x = True y = False z = x and y print(z) # True</pre>
or	At least one result in True	<pre>x = True y = False z = x or y print(z) # True</pre>
not	Results in True when expression evaluated False	<pre>x = True y = not x print(y) # False</pre>

# Task 1: Getting started

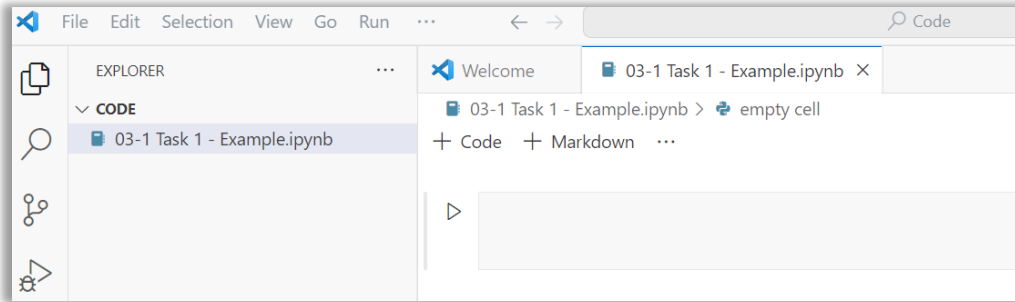
1. Setup VSCode
  - Install VSCode and Python if you have not already.
  - Create a folder for your class work outside of the downloads folder.
  - Open the folder in VSCode.
2. Download and open the 'Introduction to Python Tasks.ipynb' notebook.
  - Click play on the first box and install recommended packages.
3. Follow all the 'Task 1' sections in the notebook.

Optional: Read up on PEP8 styling for Python.



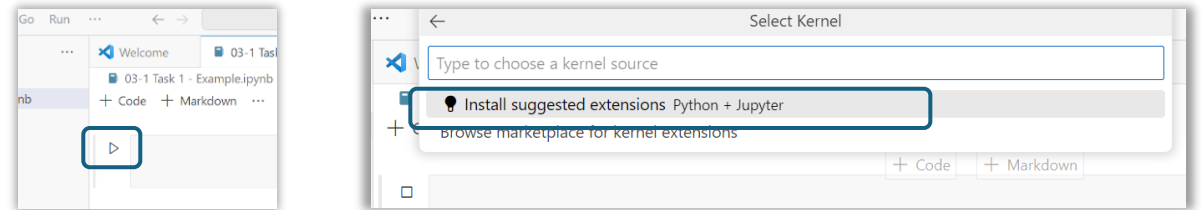
# VSCode – Running a program

## 1. Blank Notebook



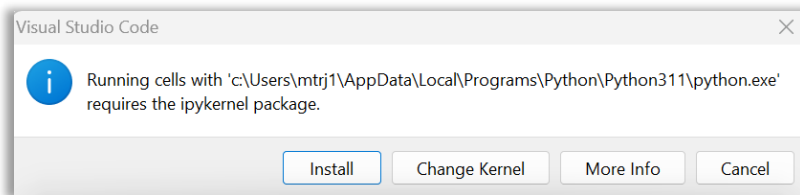
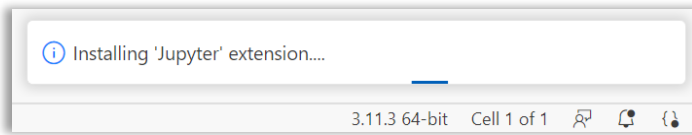
## 2. Install recommended extensions

- Press play to get prompt

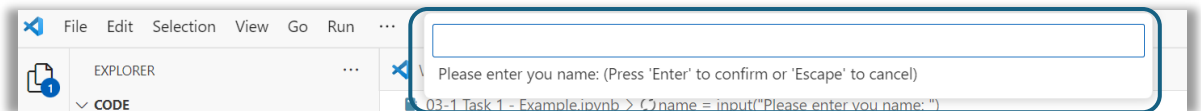
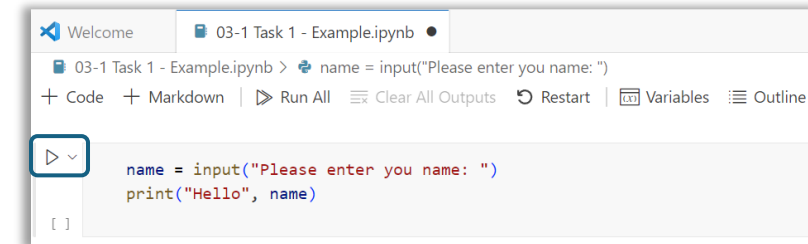


## 3. Install recommended packages

- Allow any firewall and “Trust Notebook Folder” options



## 4. Run a program – Test environment works



BREAK - 15 minutes



# Python

## Programming concepts





# Python Programming Concepts

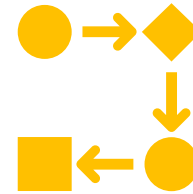
## The Three Pillars of Programming



Sequence



Conditions



Iteration

# Sequential Programming

- **Execution order**
  - Execution of code statements one after the other, in the order they were written.
- **Control structures**
  - Conditional statements - if, elif, else
  - Loops - for, while
- **Functions**
  - Organize code into reusable blocks.
- **Data types**
  - Python built-in data types - integers, floats, strings, lists and dictionaries
- **Modules**
  - Pre-written libraries - I/O, networking, data processing

What happens if the execution mode is in the wrong order?

Try it yourself!

- Can you place these instructions for washing your hands in the correct sequence?
- Scrub your hands for at least 20 seconds
- Rinse the soap from your hands under clear, running water
- Lather your hands in the soap until all areas are covered
- Wet your hands with clean, running water
- Dispense soap onto your hands

The steps and order of execution you decide here would result in an **algorithm**.



# Algorithms

Can be used to solve complex problems, following a set of specific instructions and constraints.

- **Finite sequence** of instructions.
- **Solve** a class of **specific problems** or perform a computation.
- **List of rules**
  - Constraints
- **Execution order**
  - Rules define execution order and limitations.
- **Conditionals**
  - Divert code execution.
  - Automation.

- An algorithm is a sequence of precise instructions to complete a specific task.
- Standard algorithms available:
  - Sorting a list
  - Searching a list
- Real world algorithm examples:
  - Instructions for brushing your teeth.
  - Following a recipe to bake a cake.
  - Making a hot drink, like a cup of tea.
  - Directions to a friend's house.



# Conditional Programming

- Conditional programming is a fundamental concept used in almost all programmes.
- Conditional programming is used to control the flow of a programme.
- Allows the programme to make decisions based on certain conditions.
- Actions performed dependant on the input, and the state of the programme.
- Useful for creating programmes that can handle a variety of situations.

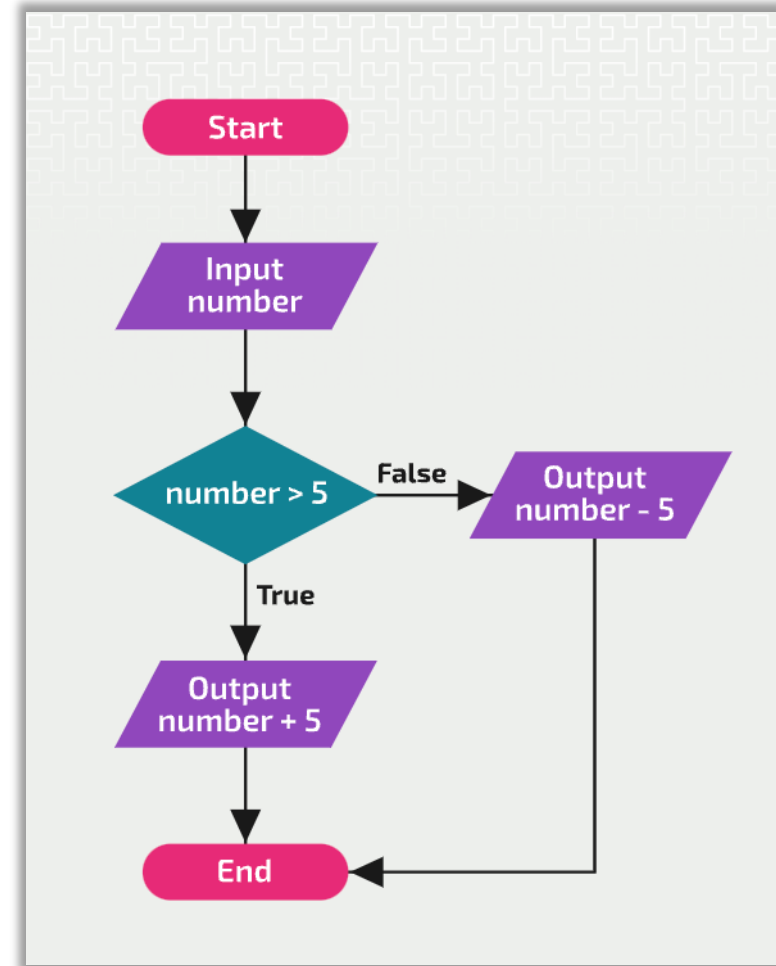
Example 1: A tax calculation program might use conditional statements to apply different tax rates.

Example 2: A login system:

- User enters wrong password:
  - Display error message.
- User enters correct password:
  - Login the user.

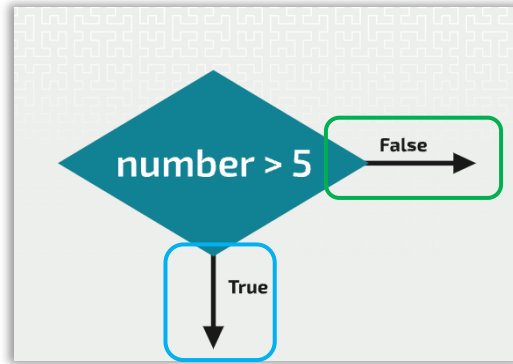
# Selection

- Selection
  - A programming concept.
  - Allows a program to choose which instructions to execute.
  - Allows more complex behaviour.
  - Controls flow of program execution.
- Control statements
  - if, elif, else.
- Evaluates condition
  - Execute different blocks of code depending on condition result.
    - **True of False.**

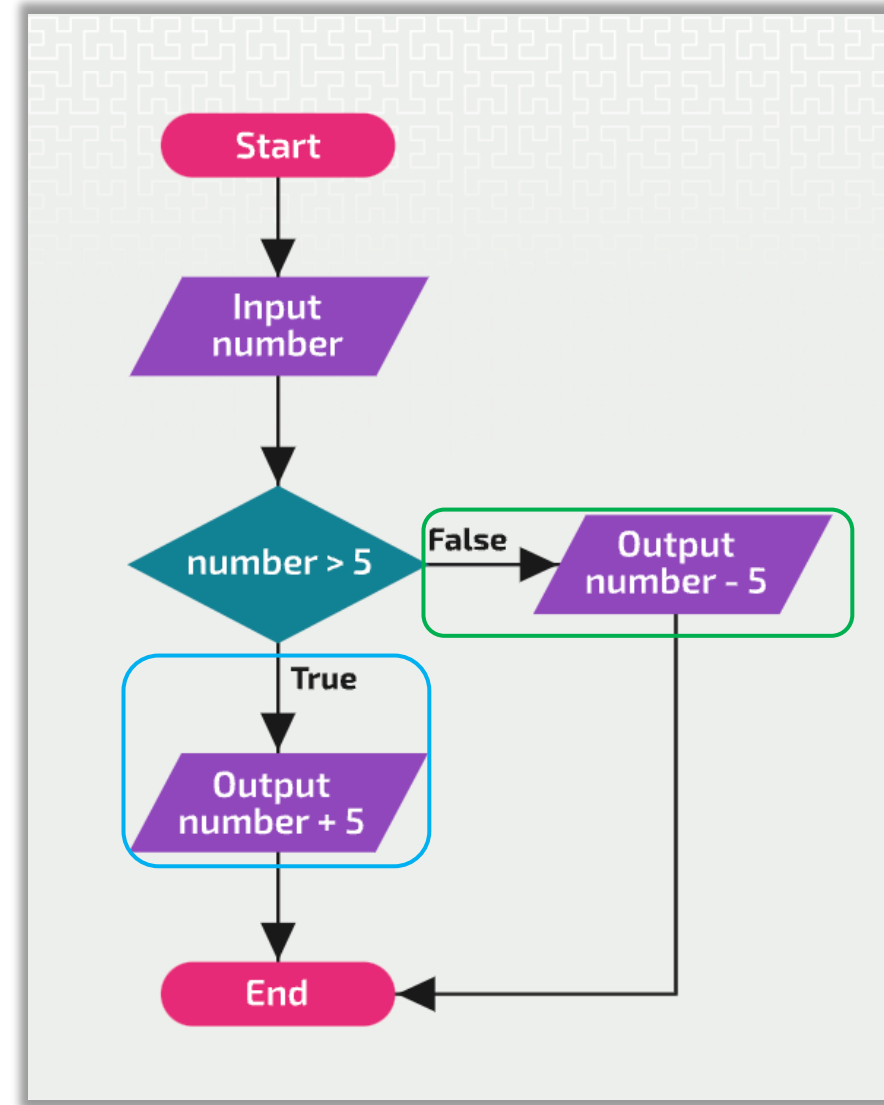


# Selection

- Conditional Evaluation

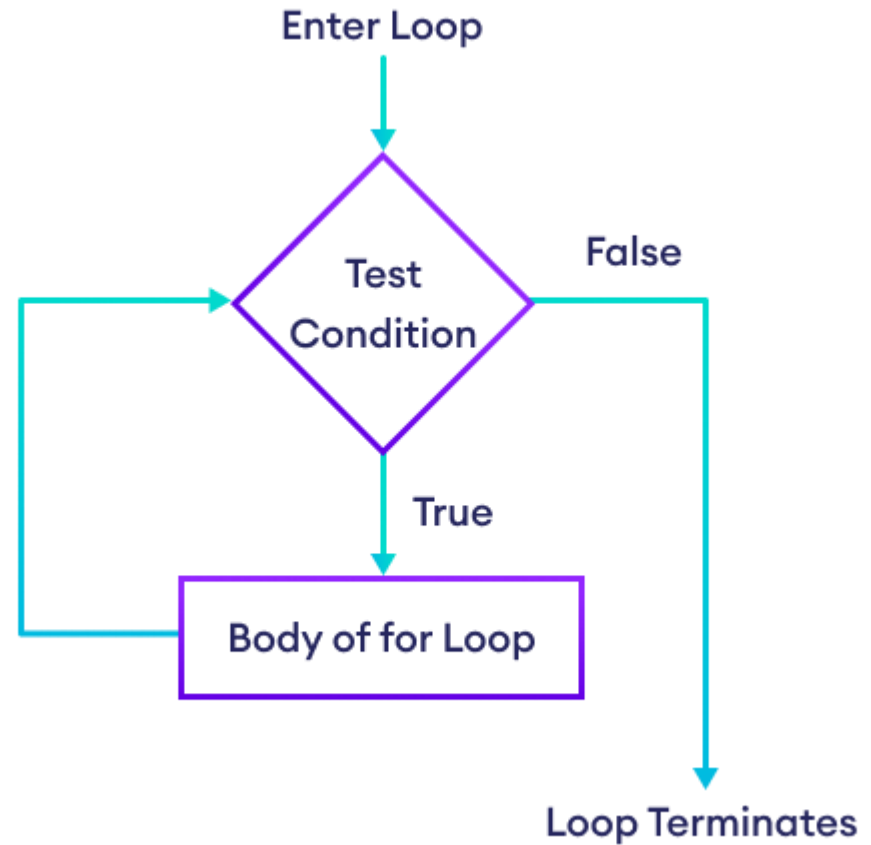


- If number > 5
  - Result = True
    - Output number + 5
- else
  - Result = False
    - Output number - 5



# Iteration

- Iteration programming is a fundamental concept used in almost all programs.
- **Iteration**
  - Repetition of a block of code.
  - Code reuse.
- **Loops**
  - *for*
    - Test Condition is a counter, e.g.  $\text{counter} < 5$ .
    - Definite iteration – a set number of loops, predefined.
  - *while*
    - Test Condition is a counter, e.g.  $\text{number} > 5$
    - Indefinite iteration. This means it's easy to cause an infinite loop if the condition is never met!



# Logical Operators

- Evaluate multiple relational expressions to return a single value.
- **Brackets**
  - Clarify the order that we want things to happen in. If there are no brackets, Python will apply the default order of operations
- **Order of operations**
  1. NOT
  2. AND
  3. OR

```
x = 5
y = 10

if x > 0 and y > 0:
    print("Both x and y are positive")

if x > 0 or y < 0:
    print("Either x is positive or y is negative (or both)")

if not x < 0:
    print("x is not negative")
```

```
x = 5
y = 10
z = -2

if (x > 0 and y > 0) or z < 0:
    print("Either both x and y are positive, or z is negative (or both)")

if not (x < 0 or y < 0):
    print("Neither x nor y is negative")
```

- $(x > 0 \text{ and } y > 0) \text{ or } z < 0$
- Is the same as
- $(x > 0 \text{ and } y > 0) \text{ or } (z < 0)$





# Comparison Operators

- Comparison Operators
- Used to compare two values
- Like logical operators

```
x = 5
y = 3

if x >= y:
    print(f"{x} is greater than or equal to {y}")
else:
    print(f"{x} is less than {y}")

if x == y:
    print(f"{x} is equal to {y}")
else:
    print(f"{x} is not equal to {y}")
```

Operator	Description	Example
<	Is less than	<b>if</b> cost < 12:
<=	Is less than or equal to	<b>if</b> cost <= 12:
>	Is greater than	<b>if</b> cost > 12:
>=	Is greater than or equal to	<b>if</b> cost >= 12:
==	Is equal to	<b>if</b> cost == 12:
!=	Is not equal to	<b>if</b> cost != 12:

# Python

## Conditional elements

# if statements

- If statements are used to test a specific condition.
  - If condition is True:
    - Execute block of code.
  - if, elif and else lines **must have a colon ':'** at the opening of a code block.
- Condition  $x > 0$  is evaluated.
- If it evaluates to **True**:
  - Block of code inside the if statement is executed.
  - "x is positive" is printed.
  - "This message is printed regardless of the value of x" is printed
- If the condition evaluates as **False**:
  - Block of code inside the if statement would be skipped.
  - "This message is printed regardless of the value of x" is still printed.

- Evaluate condition:

```
x = 5
if x > 0:
    print("x is positive")
print("This message is printed regardless of the value of x")
```

- True Output:

```
x is positive
This message is printed regardless of the value of x
```

- False Output:

```
This message is printed regardless of the value of x
```



# elif statements

- If and elif statements are used to test a specific condition.
  - If condition is True:
    - Execute block of code.
- Condition  $x > 0$  is evaluated.
- if: Evaluates to **True**
  - Block of code inside the if statement is executed.
  - "x is positive" is printed.
- if: evaluated as **False**
  - Block of code inside the if statement would be skipped.
- elif:
  - Does the same as if.
  - An extra possible condition.

- Evaluate condition:

```
x = 0
if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
```

- If  $x > 0$ 
  - Output:
- Elif  $x < 0$ 
  - Output:

```
x is positive
```

```
x is negative
```

# else statements

- else statements are used with an if condition.
  - If condition is False:
    - Execute else instead.
  - **else is only executed when an “if” condition is False.**
- Condition  $x > 0$  is evaluated.
- if: Evaluates to **True**
  - Block of code inside the if statement is executed.
  - “x is positive” is printed.
- if: evaluated as **False**
  - Block of code inside the if statement would be skipped.
- elif:
  - Does the same as if.
  - An extra possible condition.

- Evaluate condition:

```
x = 0
if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is zero")
```

- If  $x > 0$ 
  - Output:
- Elif  $x < 0$ 
  - Output:
- Else
  - Output:

x is positive

x is negative

x is zero

# elif Larger Example

- The program uses a series of if and elif statements to check the value of x.
- x (85) represents a student's score on a test.
- x is greater than or equal to 80 but less than 90.
- condition in the second elif statement is True.
- block of code inside that statement is executed and "Grade: B" is printed.
- What would be printed out when:
  - x was greater than or equal to 90?
  - x was less than 60?

- Evaluate condition:

```
x = 85
if x >= 90:
    print("Grade: A")
elif x >= 80:
    print("Grade: B")
elif x >= 70:
    print("Grade: C")
elif x >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

- elif x >= 80
  - Output:

Grade: B



# Nested Conditionals

- Nested if statements can check multiple conditions
- Outer if:
  - Condition  $x > 0$  is evaluated
  - When True
    - Inner if:
      - Condition  $y > 0$  is evaluated
        - If True ( $y > 0$ )
          - Prints "Both x and y are positive"
        - If False ( $y < 0$ )
          - Prints "x is positive and y is not positive"
    - When False
      - Outer else:
        - Prints "x is not positive"

```
x = 5
y = 10

if x > 0:
    if y > 0:
        print("Both x and y are positive")
    else:
        print("x is positive and y is not positive")
else:
    print("x is not positive")
```

```
x = 5
y = 10

if x > 0:
    if y > 0:
        print("Both x and y are positive")
    else:
        print("x is positive and y is not positive")
else:
    print("x is not positive")
```

# Case switching — before Python 3.10

- Many programming languages include a “switch” keyword
  - Alternative method multiple conditionals
  - Also known as: match, select or inspect
- Python does not have a built-in keyword before version 3.10
- Can imitate using if, elif and else
- What would be printed out when:
  - 1. If error\_code was equal to '403'?
  - 2. If error\_code was equal to 'timeout'?

```
error_code = 400

if error_code == 400:
    print("Bad Request")
elif error_code == 401:
    print("Unauthorized")
elif error_code == 403:
    print("Forbidden")
elif error_code == 404:
    print("Page Not Found")
elif error_code == "timeout":
    print("Timeout Error")
else:
    print("Unknown Error")
```

- Output:

Bad Request





# Case switching

- The program is imitating case switching using if, elif and else to check error\_code.
- What would be printed out when:
  - error\_code was equal to '403'?
    - Forbidden
  - 2. error\_code was equal to 'timeout'?
    - Timeout Error

```
error_code = 400

if error_code == 400:
    print("Bad Request")
elif error_code == 401:
    print("Unauthorized")
elif error_code == 403:
    print("Forbidden")
elif error_code == 404:
    print("Page Not Found")
elif error_code == "timeout":
    print("Timeout Error")
else:
    print("Unknown Error")
```

# Case Switching — Python 3.10+

- Python 3.10 introduced Structural Pattern Matching.
- Match and case keywords now available.
- Like if, elif, else.
- Results in cleaner code.
- Variable “error\_code” only entered once.
- ‘\_’ wildcard if no case matches.
  - Performs the same function as else

```
error_code = 400

match error_code:
    case 400:
        print("Bad Request")
    case 401:
        print("Unauthorized")
    case 403:
        print("Forbidden")
    case 404:
        print("Page Not Found")
    case "timeout":
        print("Timeout Error")
    case _:
        print("Unknown Error")
```

- Output:

Bad Request



## Task 2: Selection

1. Return to the 'Introduction to Python.ipynb' notebook.
2. Follow all the 'Task 2' sections in the notebook

Optional: Read up on f strings and implement an example



# Python

## Data structures



# Data Structures

Data Type	Description	Example
List	A list object is an ordered collection of one or more items of data, this does not need to be all the same type, but must be put in square brackets.	<pre>my_list = [1, "apple", 3.14]</pre>
Tuple	A Tuple object is an ordered collection of one or more items of data, this does not need to be of the same type but must be put inside parentheses.	<pre>my_tuple = (1, "apple", 3.14)</pre>
Set	A set is a collection of items that is unordered and unindexed. Sets do not allow duplicate items	<pre>my_set = {1, "apple", 3.14}</pre>
Dictionary	A dictionary object is an unordered collection of data in a key:value pair form. A collection of such pairs is enclosed in curly brackets.	<pre>my_dict = {"name": "John", "age": 30}</pre>



# Data Structures

- Data structures are a way of organizing and storing data in a computer.
- They allow data to be accessed and used efficiently.
- Different types of data structures are suited to different kinds of applications.
- Common data structures include arrays, lists, stacks, queues, trees, graphs, and hash tables.
- Each data structure has its own advantages and disadvantages (this could be a whole course by itself!)

```
# List
my_list = [1, "apple", 3.14]
my_list.append("banana")
print(my_list) # [1, "apple", 3.14, "banana"]
```

```
# Tuple
my_tuple = (1, "apple", 3.14)
print(my_tuple[1]) # "apple"
```

```
# Set
my_set = {1, "apple", 3.14}
my_set.add("banana")
print(my_set) # {1, 3.14, 'banana', 'apple'}
```

```
# Dictionary
my_dict = {"name": "John", "age": 30}
my_dict["age"] = 40
print(my_dict) # {'name': 'John', 'age': 40}
```



# Data Structures

- Large amounts of data need to be organised often on computer storage.
- Data can be organised into specific structures, like files and folders on a computer.
- Programmers need to consider how data is stored in programs to access and manipulate it.
- Data structures can be thought of as containers that store collections of data.

```
# List
my_list = [1, "apple", 3.14]
my_list.append("banana")
print(my_list) # [1, "apple", 3.14, "banana"]
```

```
# Tuple
my_tuple = (1, "apple", 3.14)
print(my_tuple[1]) # "apple"
```

```
# Set
my_set = {1, "apple", 3.14}
my_set.add("banana")
print(my_set) # {1, 3.14, 'banana', 'apple'}
```

```
# Dictionary
my_dict = {"name": "John", "age": 30}
my_dict["age"] = 40
print(my_dict) # {'name': 'John', 'age': 40}
```

# Data Structures

- Primitive data types store single values.
  - Integer, boolean, etc.
- Complex data structures allow for more sophisticated organization and manipulation of data.
  - Lists, dictionaries.
- Static and dynamic data structures.
  - Stacks and queues provide methods;
    - Adding,
    - Removing,
    - Traversing data.

```
# Primitive data type
x = 5 # integer
y = True # boolean

# Complex data structure (list)
numbers = [1, 2, 3, 4, 5]

# Static data structure (tuple)
weekdays = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

# Dynamic data structure (dictionary)
contacts = {
    "John": "555-1234",
    "Jane": "555-5678"
}
```

More information on stacks and queues:

<https://www.geeksforgeeks.org/static-data-structure-vs-dynamic-data-structure/>





# Lists

- Lists are a built-in, ordered, and mutable data type in Python.
- Mutable means we can change its content without changing its identity.
- Can store items of different data types, including other lists.
- Lists are created using square brackets [] or the list() constructor.
- Common operations include:
  - Indexing.
  - Slicing.
  - Appending.
  - Inserting.
  - Removing.
  - Sorting.

```
# Create a list
my_list = [1, 2, 3, 4, 5]

# Access elements by index
first_element = my_list[0]
last_element = my_list[-1]

# Add an element to the end of the list
my_list.append(6)

# Insert an element at a specific position
my_list.insert(0, 0)

# Remove an element from the list
my_list.remove(3)

# Remove an element by index
del my_list[1]
```

```
# Get the length of the list
length = len(my_list)

# Sort the list in ascending order
my_list.sort()

# Reverse the order of the list
my_list.reverse()

# Iterate over the elements in the list
for element in my_list:
    print(element)
```



# Tuples

- Tuples are a built-in, ordered, and **immutable** data type.
- Can store items of different data types.
- Tuples are created using parentheses () or the tuple() constructor.
- Common operations include:
  - Indexing.
  - Slicing.

```
# Create a tuple
my_tuple = (1, 2, 3, 4, 5)

# Access elements by index
first_element = my_tuple[0]
last_element = my_tuple[-1]

# Tuples are immutable, so you cannot add or remove elements
# However, you can create a new tuple with the desired elements
my_tuple = my_tuple + (6,)

# Get the length of the tuple
length = len(my_tuple)

# Iterate over the elements in the tuple
for element in my_tuple:
    print(element)
```



# Sets

- Sets are a built-in, unordered, and mutable data type.
- Can store items of different data types.
- Unique values only.
- Sets are created using curly brackets {}.
- Common operations include:
  - Adding.
  - Removing.
  - Checking for membership.

```
# Create a set
my_set = {1, 2, 3, 4, 5}

# Add an element to the set
my_set.add(6)

# Remove an element from the set
my_set.remove(3)

# Check if an element is in the set
is_member = 4 in my_set

# Get the length of the set
length = len(my_set)

# Iterate over the elements in the set
for element in my_set:
    print(element)
```



# Dictionaries

- Dictionaries are a built-in, ordered, and mutable data type.
- Store key-value pairs.
  - Must be unique and hashable.
  - Hash type would never change.
- Dictionaries are created using curly brackets {} or the dict() constructor.
- Common operations include:
  - Adding.
  - Removing.
  - Accessing elements by key.

```
# Create a dictionary
my_dict = {"one": 1, "two": 2, "three": 3}

# Add a key-value pair to the dictionary
my_dict["four"] = 4

# Remove a key-value pair from the dictionary
del my_dict["one"]

# Access an element by key
value = my_dict["two"]

# Get the length of the dictionary
length = len(my_dict)

# Iterate over the keys in the dictionary
for key in my_dict:
    print(key)
```



# Task 3: Data Structures

1. Return to the 'Introduction to Python Tasks.ipynb' notebook.
2. Follow all the 'Task 3' sections in the notebook.

Optional: Read up on slicing and implement an example.





# LUNCH - 30 minutes



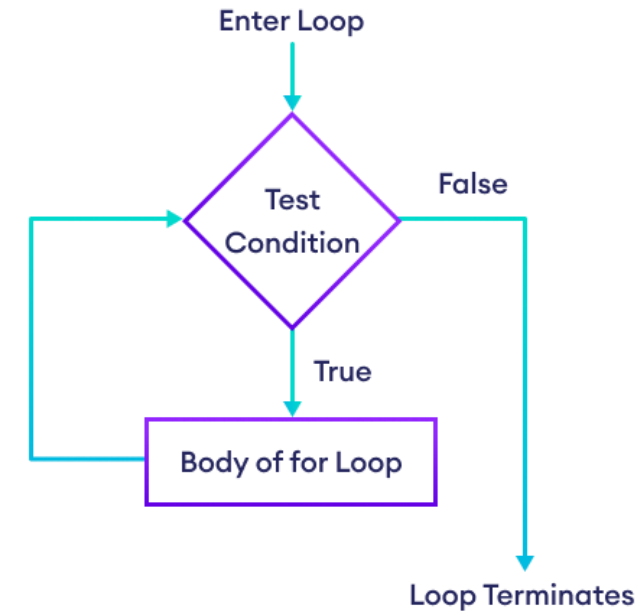
# Python

## Iteration



# Iteration

- Iteration refers to repeating a step;
  - Looping through a code block until a condition is matched,
  - Traversing elements of a collection one by one.
- Python loops
  - For
    - Iterates over elements.
      - A counter.
      - A group of elements.
  - While
    - Loops until exit condition matched.
    - Easy to create infinite loop.



```
# Create a variable to store whether we have arrived
arrived = False

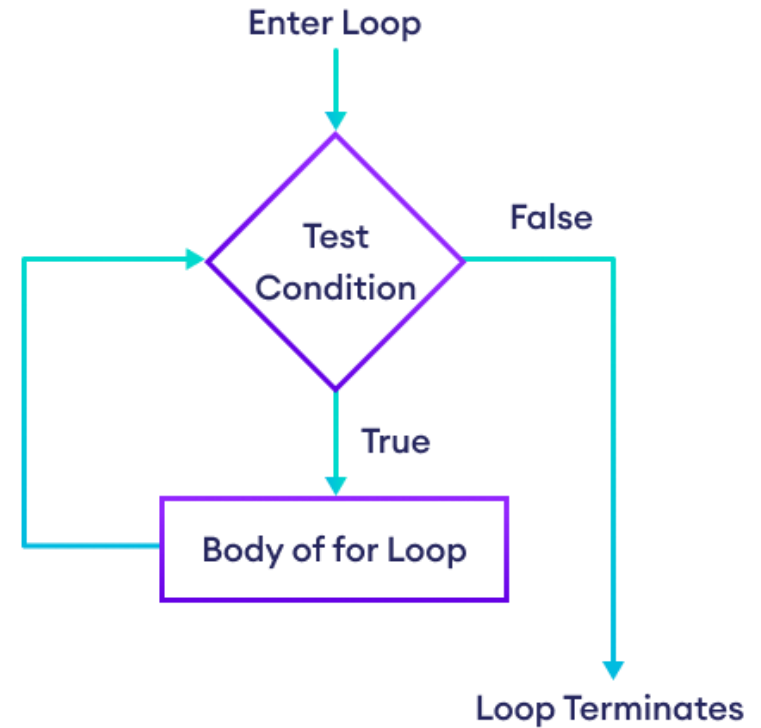
# Iterate using a while loop
while not arrived:
    print("Are we there yet?")
    arrived = True

print("We have arrived!")
```



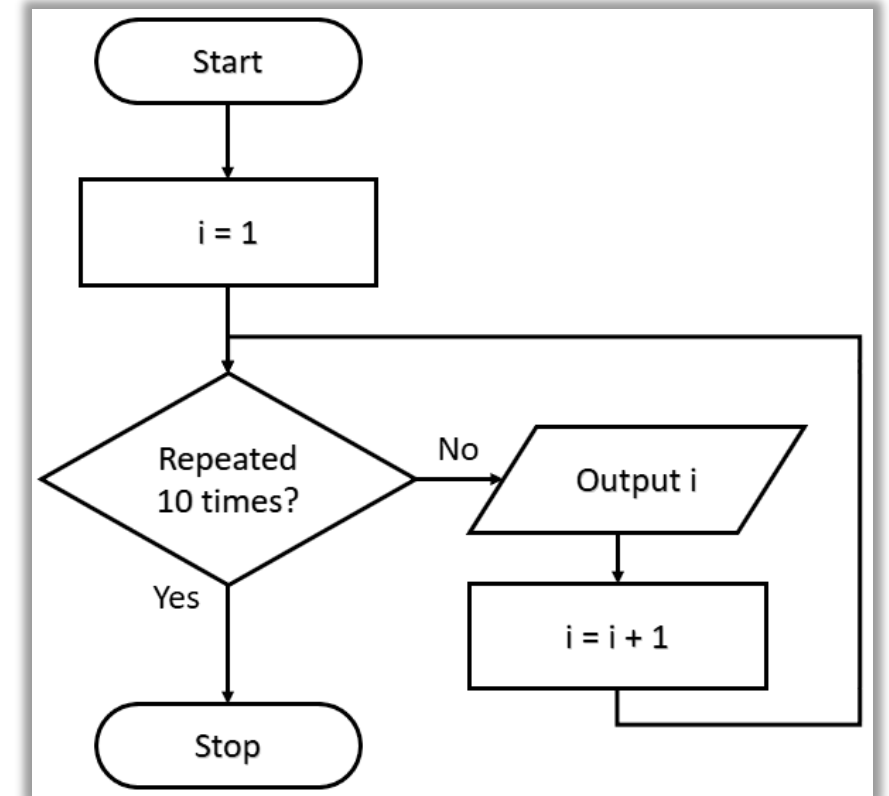
# Iteration

- There are 4 main types of loops:
  - For,
  - While,
  - Do while,
  - Repeat until.
- Python does **not** include:
  - Do While,
  - Repeat until.



# For Loop

- For loop will repeat a set amount of times.
  - Count controlled.
  - Specify the number of times to repeat.
  - Can be number of elements in a list.
  - Sequence is stepped through one-by-one.
- Example:
  - Outputs: 1, 2, 3, 4, 5, 6, 7, 8 and 9
  - When it gets to 10 the loop would stop.
    - 10 would not be printed.



```
for i in range(1, 10):  
    print(i)
```

# For Loop

- Step can be defined in loop header.
- range() function generates a sequence of numbers.
- Start, stop, step value.
- For loop iterates over the sequence in range.
- Code inside loop is executed for each number in the sequence.
- Example:
  - Outputs: 0, 2, 4, 6, 8
  - Remember that the stopping value of the range function is **exclusive**.
    - When it gets to 10 the loop would stop.
    - 10 would not be printed.

```
# Iterate over a range of numbers from 0 to 10 with a step of 2
for i in range(0, 10, 2):
    print(i)
```



# For Loop

- For loop can iterate over the elements of a collection.
  - This is known as a foreach loop in other languages.

```
# Create a list of fruits
fruits = ["apple", "banana", "cherry"]

# Iterate over the elements of the list using a for loop
for fruit in fruits:
    print(fruit)
```

- Example:
  - Each iteration
    - fruit takes on the next value in the list.
    - code executed with new fruit value.
  - Continues until all list elements have been traversed.



- Example of for loop without using 'for' and 'in'
- More code needed
- Less readable

```
# Create a list of fruits
fruits = ["apple", "banana", "cherry"]

# Get the length of the list
n = len(fruits)

# Iterate over the elements of the list using a for loop and indexing
for i in range(n):
    fruit = fruits[i]
    print(fruit)
```

- Output for both examples

```
apple
banana
cherry
```

# For Loop Example

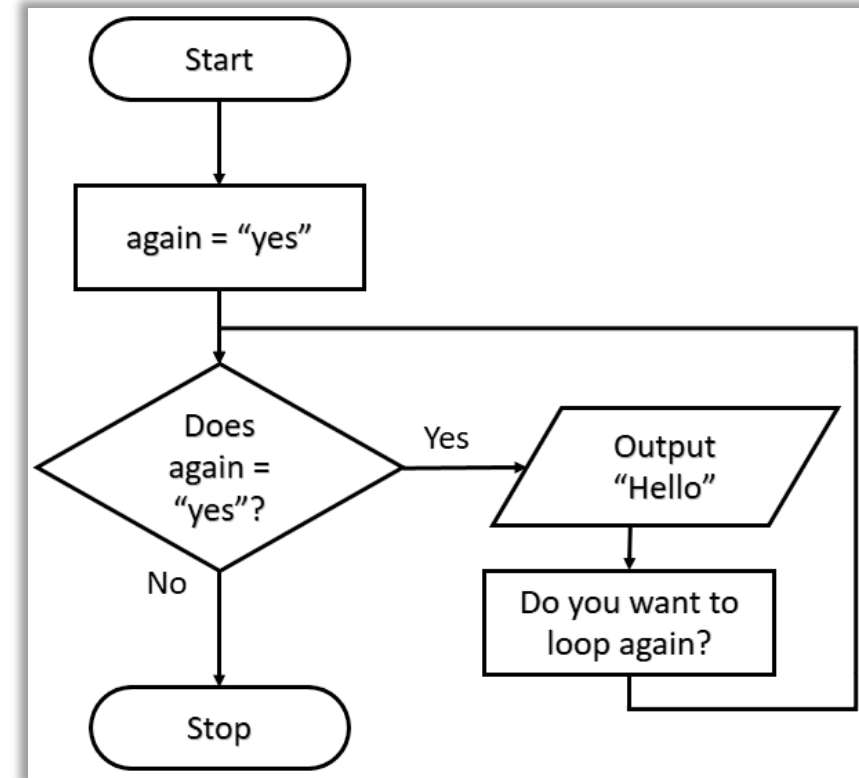
1. What is the purpose of the rainbow variable?
2. What type of data structure is rainbow?
3. How many elements are in the rainbow list?
4. What does the for loop do in this code?
5. What is the output of this code?
6. How could this code be modified to print the colours in reverse order?

```
rainbow = ["red", "orange", "yellow", "green", "blue", "indigo", "violet"]  
  
for colour in rainbow:  
    print(colour)
```



# While Loop

- A while loop will loop until exit condition matched.
- Also known as, a flag, rogue value or sentinel.
- Condition checked before code is run.
  - If condition is True:
    - While loop runs,
      - If 'again' equals 'yes' (==)
      - Keeps running while condition is True.
  - If condition is False:
    - While loop skipped.



```
again = "yes"

while again == "yes":
    print("Hello")
    again = input("Do you want to loop again? ")
```

# While Loop

- Condition checked before code is run.
  - If “Bob” **is not** entered,
    - While loop runs.
      - When not equal (!=)
      - Asks the user to enter your name again.
    - Keeps running while condition is True.
  - If “Bob” **is** entered,
    - While loop exits.
  - Prints “Hi, Bob!”.

```
print("Please enter your name: ")
name = input()

while name != "Bob":
    print("Try again, Bob")
    print("Please enter your name: ")
    name = input()

print("Hi, Bob!")
```

What would happen if the first line of code was not there?

# While Loop Example

- 1. How many times will this code block loop?

```
total = 0
count = 0
while total <= 100:
    total = total + 1

print("The total is", total)
```

- 2. How many times will this code block loop?

```
total = 0
count = 0
while total < 100:
    total = total + 1

print("The total is", total)
```

Note: Adding a counter in this manner, imitates a for loop



# While Loop Example - Answers

1. How many times will this code block loop?

```
total = 0
count = 0
while total <= 100:
    total = total + 1

print("The total is", total)
```

101 times

- Counter includes zero
- Counter **includes** "100"
  - Less than or equal to 100

2. How many times will this code block loop?

```
total = 0
count = 0
while total < 100:
    total = total + 1

print("The total is", total)
```

100 times

- Counter includes zero
- Counter **does not include** "100"
  - Less than 100

# Infinite Loop

- While
  - Loops until exit condition matched
  - Easy to create infinite loop
- Condition controlled
- Condition never matched
  - Infinite Loop

```
# Create a variable to store whether we have arrived
arrived = False

# Iterate using a while loop
while not arrived:
    print("Are we there yet?")
```

# Infinite Loop Solved

- Infinite while loop
- No ability to change the exit condition
  - arrival variable
- Will continue until program force closed

```
# Create a variable to store whether we have arrived
arrived = False

# Iterate using a while loop
while not arrived:
    print("Are we there yet?")
```

- Finite while loop
- User input utilised to change exit condition
  - arrival variable
- Exits loop when “y” for arrival

```
# Create a variable to store whether we have arrived
arrived = False

# Iterate using a while loop
while not arrived:
    print("Are we there yet?")
    # Check if we have arrived using an if condition
    if input("Have we arrived? (y/n): ") == "y":
        arrived = True

print("We have arrived!")
```



# Nested Loops

- A nested loop is a loop that is inside another loop.
- Inner loop is executed each iteration of the outer loop.
- Inner loop can use the variables of the outer loop. This is called scope.
- Use case: iterate over multiple dimensions of data, such as a two-dimensional list.
- Order of nested loops can affect order of data processed.
- Can quickly become computationally expensive.

```
# Create a two-dimensional list of numbers
numbers = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Iterate over the rows of the list using an outer for loop
for row in numbers:
    # Iterate over the elements of each row using an inner for loop
    for number in row:
        print(number)
```

1  
2  
3  
4  
5  
6  
7  
8  
9



# Nested Loops

- A nested for loop to generate and print the multiplication table.
  - Numbers 1 to 3
- The outer for loop iterates over the numbers 1 to 3.
  - Repeats 3 times.
  - Remember, range() is exclusive.
- The inner loop iterates 10 times.
  - For each iteration of the outer loop.
- The inner loop repeats 30 times in total.
  - 3 (outer) \* 10 (inner).

```
for num in range(1,4):  
    for multiplier in range(1,11):  
        result = num * multiplier  
        print(f"{num} * {multiplier} = {result}")
```

- Output (truncated)

```
1 * 1 = 1  
1 * 2 = 2  
1 * 3 = 3  
...  
3 * 8 = 24  
3 * 9 = 27  
3 * 10 = 30
```



# Nested Loops

- Multiple different loops can be nested together.
  - Remember, `int()` type casts the inputted string.
  - Can now be used for comparison.

```
# Create a list of numbers for the user to guess
numbers_to_guess = [1,4,8,3,10]

# Print instructions for the user
print("Guess my numbers, each number is between 1 and 10")

# Iterate over the numbers in the list using a for loop
for number in numbers_to_guess:
    # Prompt the user to enter a guess
    guess = int(input("Enter a number to guess: "))

    # Continue prompting the user for guesses until they enter the correct number
    while guess != number:
        guess = int(input("Enter a number to guess: "))

    # Congratulate the user for correctly guessing the current number
    print("Well done, Please enter the next number to guess")
```

## Output

```
Guess my numbers, each number is between 1 and 10
Enter a number to guess: 5
Enter a number to guess: 1
Well done, Please enter the next number to guess
Enter a number to guess: 4
Well done, Please enter the next number to guess
Enter a number to guess: 8
Well done, Please enter the next number to guess
Enter a number to guess: 3
Well done, Please enter the next number to guess
Enter a number to guess: 10
Well done, Please enter the next number to guess
```



# Task 4: Loops

1. Return to the 'Introduction to Python Tasks.ipynb' notebook
2. Follow all the 'Task 4' sections in the notebook



# Python

## Subroutines, procedures and functions



# Subroutine

- A subroutine is a sequence of program instructions that performs a specific task.
- Packaged as a unit.
- Known as procedures, functions, or methods.
  - **Function** or **procedure** in Python.
- Break down a complex program into smaller, more manageable parts.
- Abstraction
  - Details of how a subroutine works do not need to be known once written initially.

- Can be called from other parts of the program
  - Code reuse
- Take input in the form of arguments
- Return output in the form of a return value
- Code more readable and easier to maintain

```
# Define a function that adds two numbers
def add(a, b):
    return a + b

# Call the add function and print the result
print(add(3, 4)) # 7
```



# Procedures

- A procedure is a named block of code that performs a specific task.
- Print() is a common example. No need to understand how print() works, just know it will print to the screen.
- Procedures do not return any value.
- Procedures can be called by another part of a program.

## While loop with print

```
print("Please enter your name: ")
name = input()

while name != "Bob":
    print("Try again, Bob")
    print("Please enter your name: ")
    name = input()

print("Hi, Bob!")
```

## For loop output from print

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
...
3 * 8 = 24
3 * 9 = 27
3 * 10 = 30
```

# Procedures

- Procedures can be used to display menu options.
- Can be called anywhere the menu is needed.
- Call using display\_menu()
- What if you needed to change the menu options?
  - Only need to change in this procedure, rather than everywhere the menu is needed.
  - Less error prone.

```
def display_menu():  
    print("1: Log in")  
    print("2: Sign up")  
    print("9: Quit")
```

# Functions

- A function is defined using:
  - **def** keyword,
  - Followed by the function name,
  - Arguments in parentheses.

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

- Colon (:) after the arguments
  - Start of a new block of code
  - **Must** be present
    - Automatic indentation benefit

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

- Return keyword to specify the output from function

```
    return a + b
```

- The function is called by its name, followed by parentheses containing any arguments.
- The add function takes two arguments 'a' and 'b' and returns their sum.

```
# Define a function that adds two numbers  
def add(a, b):  
    return a + b
```

- We call the add function with arguments 3 and 4, and it returns 7.

```
# Call the add function and print the result  
print(add(3, 4)) # 7
```



# Function

- Main function runs first, then calls the calculate\_area\_of\_square function, like a procedure.
- Need to define (def) a function before we call it. Interpreted languages need to know function definition.
- Main function sets the value of the length variable and calls the calculate\_area\_of\_square function.
- Result of calling the calculate\_area\_of\_square function is stored in the result.
- Prints out the result.

```
# Define a function to calculate the area of a square
def calculate_area_of_square(length):
    area = length * length
    return area

# Define the main function
def main():
    # Set the value of the length variable
    length = 12
    # Call the calculate_area_of_square function and store the result in the result
    variable
    result = calculate_area_of_square(length)
    # Print the result
    print(result)

# Call the main function
main() # 144
```



# Procedures vs Functions

- Procedures and functions can be seen as interchangeable.
- Industry does not always differentiate.
- A function
  - Usually used for:
    - Input,
    - Performing a calculation,
    - Returning one or more values.
- A procedure:
  - Used to create a function that returns no value,
  - Displaying information to the screen.

- Function

```
# Define a function to calculate the area of a square
def calculate_area_of_square(length):
    area = length * length
    return area
```

- For loop output from print

```
def display_menu():
    print("1: Log in")
    print("2: Sign up")
    print("9: Quit")
```

# Parameters

- Data to pass to a subroutine.
- Parameters are treated as local variables and can only be seen inside the function.
- Parameters are the preferred way to share data and variables within a program.
- Global variables can be seen by the whole program.
  - Not good practice.
  - Increases the risk of problems arising.
  - Can be modified by other parts of the program.
    - Values might not be as expected.

- Procedure with no parameters

```
def display_menu():
```

- Procedure with string parameter

```
print("1: Log in")  
print("2: Sign up")  
print("9: Quit")
```

# Arguments

- Parameters passed to subroutine when it is called.
- Arguments are the parameters set for a procedure or function to accept.
- E.g. user is asked to input their name.
- user\_name is passed into welcome\_user and becomes 'user' argument inside welcome\_user procedure.
- The procedure arguments do not need to be the same name, but we will need to use the procedure's argument name inside the procedure.

```
# Define a function that welcomes the user
def welcome_user(user):
    print("Greetings " + user)

# Define the main function
def main():
    # Prompt the user to enter their name
    user_name = input("What is your name?")
    # Call the welcome_user function with the user's name as an argument
    welcome_user(user_name)
```

Note: like **function** and **procedure** are used interchangeably, **argument** and **parameter** are also used interchangeably





# Arguments

- Number of arguments must match the number of parameters.
  - There are cases where arguments could be optional.
- Some languages allow a default value for arguments.
  - Python allows default arguments.
  - Argument is still set automatically.

- 'name' does not have a default set and must be passed in when subroutine is called.
- 'greeting' has 'Hello' as a default value, so when no parameter is passed, greeting = 'Hello'.
- When parameter is passed, greeting = passed parameter.

```
def greet(name, greeting='Hello'):
    return f'{greeting}, {name}!'
```

# Multiple parameters

- When there are multiple parameters, the parameters are separated by a comma.

```
def volume(height, width, depth):
```

- Order of the arguments is the order parameters are passed in.
  - If number of parameters do not match number of arguments, we will encounter an error unless they have default values.
- Calculate\_volume is defined with three parameters:
  - Height, width and depth.
- The parameters are passed into the function arguments.

```
result = volume(height, width, depth)
```

Multiple parameters to match arguments.

```
def volume(height, width, depth):  
    return height * width * depth  
  
height = float(input('Enter the height: '))  
width = float(input('Enter the width: '))  
depth = float(input('Enter the depth: '))  
  
result = volume(height, width, depth)  
print(f'The volume of the box is {result}')
```

Output

```
Enter the height: 2  
Enter the width: 3  
Enter the depth: 4  
The volume of the box is 24.0
```

## Task 5: Subroutines

1. Return to the 'Introduction to Python Tasks.ipynb' notebook.
2. Follow all the 'Task 5' sections in the notebook.





## Task 6: Random

1. Return to the 'Introduction to Python Tasks.ipynb' notebook.
2. Follow all the 'Task 6' sections in the notebook.





# Suggested further reading

PEP8 style guide

<https://peps.python.org/pep-0008/>

Practical Python learning

<https://learnpythonthehardway.org/>

Object Oriented Programming (OOP) concepts

<https://www.geeksforgeeks.org/python-oops-concepts/>

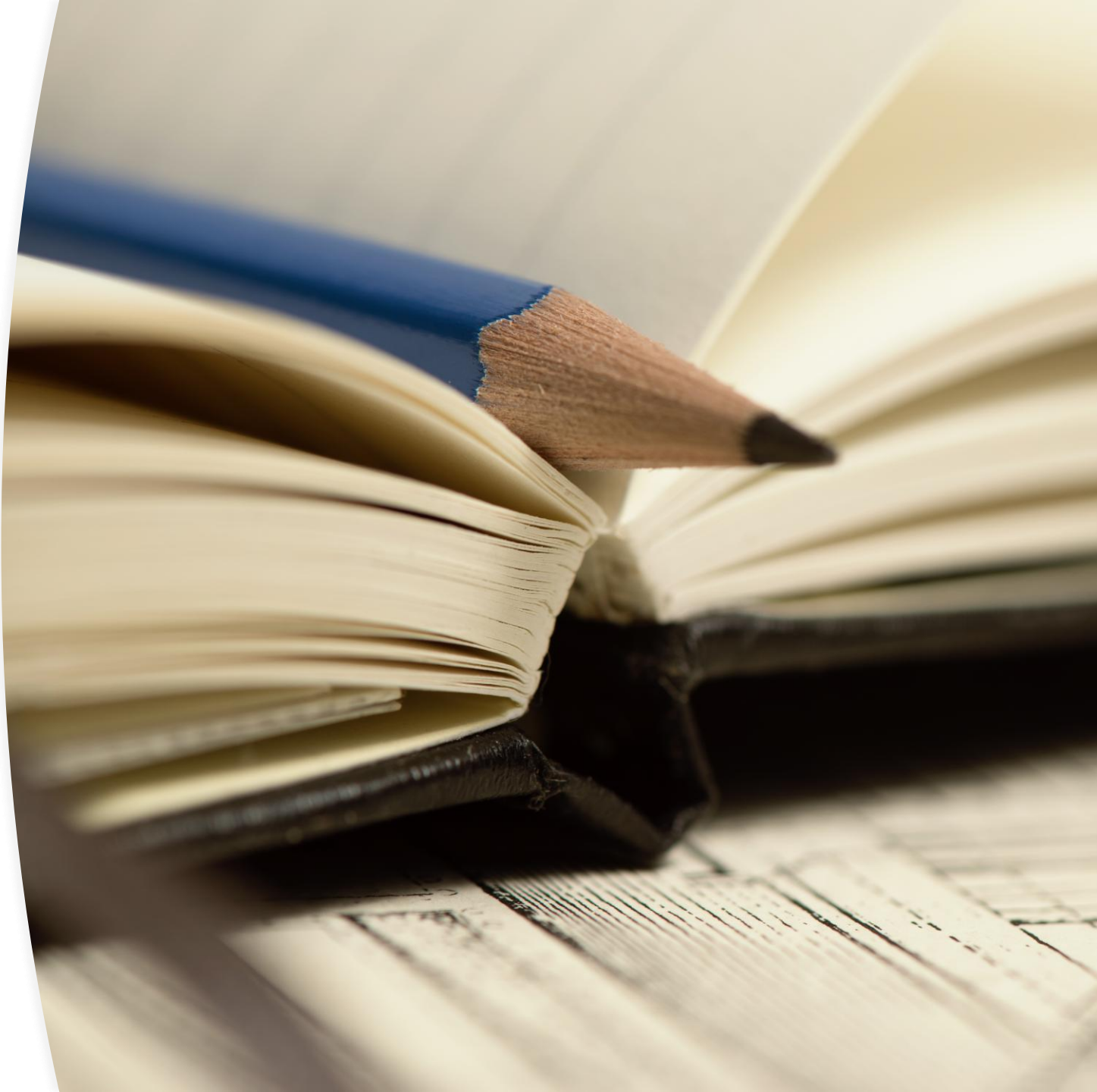
Parsing text and web scraping (Beautiful Soup)

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>



# Learner journal (2 minutes)

- What was the highlight of my week?  
What was meaningful for me?
- What is one thing I can do to set myself up for success next week?
- What new skill or idea excited me the most this week, and how can I apply it further?
- What strategies or tools helped me to stay focused and motivated?
- What am I now curious about?





# Thank you!

## Any questions?



# Next session:

## Introduction to Python Pandas