3. Python Programming Fundamentals

Overview

1. Example application of the week
2. Recap of last time
3. Conditions and branching
4. Loops
5. Functions

Example application: AlphaGo

Recap: python in jupyter

- Jupyter is a programming environment where snippets of code can be run piece by piece.
- The print function can be used to examine the results of your code.

```
In[1]:  x = 5
        y = 4
        print(x + y)
```

```
Out[1]: 9
```

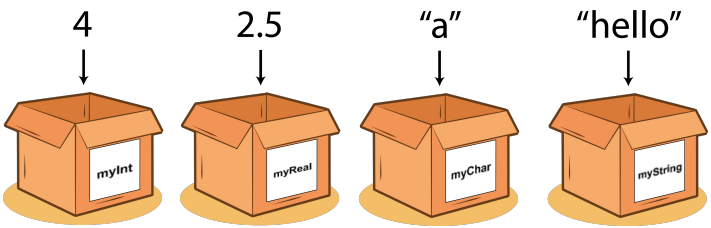
- Jupyter will also always display the last thing in a cell

```
In[2]:  x = 5
        y = 4
        x + y
```

```
Out[2]: 9
```

Recap: variables

- Variables are named containers where data of different types can be stored.



- Variables are assigned using the `=` symbol

```
name = 'Femi'     # name is a variable with value 'Femi'
age = 28          # age is a variable with value 28
```

Recap: integers and floats

- Integers (`int`) and floats (`float`) are both types of number.
- `int`s are whole numbers, and `float`s are any number with a decimal point.

```
In[3]:  x = 6
        y = 5.42
        print(type(x), type(y))
```

```
Out[3]: <class 'int'> <class 'float'>
```

- All the normal mathematical things can be done with these types.

```
In[4]:  print(x + y, x - y, x * y, x / y)
```

```
Out[4]: 11.42, 0.58, 32.512, 1.1070110701107012
```

Recap: strings

- Strings are pieces of text.
- When assigning a string, it must be enclosed in quotation marks.

```
my_text = 'hello, nice to meet you'   # right!
my_text = hello, nice to meet you     # wrong!
```

- A sub-string can be accessed by *indexing*.

```
        # index:  |01234567....
In[4]:  my_text = 'hello, nice to meet you'
        print(my_text[7:14])
```

```
Out[4]: 'nice to'
```

Recap: lists

- Lists are used for storing a sequence containing multiple pieces of data.

```
my_list = [42, 'dog', -0.5, 'cat', ['fish', 'whale', 'dolphin']]
```

- Lists have a length, which can be found using the `len()` function, e.g. `print(len(my_list)` .
- Individual items or sub-lists can be accessed by indexing.

```
In[5]:  print(my_list[1])
        print(my_list[2:4])
```

```
Out[5]: 'dog'
        [-0.5, 'cat']
```

Recap: lists

- Lists can be added together ('*concatenated*').

```
In[6]:  shopping1 = ['eggs', 'milk', 'apples']
        shopping2 = ['beans', 'pizza', 'rice']
        print(shopping1 + shopping2)
```

```
Out[6]: ['eggs', 'milk', 'apples', 'beans', 'pizza', 'rice']
```

- A new item can be added onto the end of a list by using `.append()`

```
In[7]:  squad = ['Kane', 'Sterling', 'Rashford']
        squad.append('Sancho')
        print(squad)
```

```
Out[7]: ['Kane', 'Sterling', 'Rashford', 'Sancho']
```

New topic: conditions

Conditions

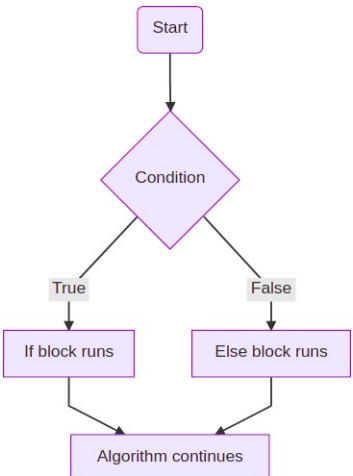
- When writing an algorithm, it is often necessary to include a condition determine what path is taken next.
- In Python this is done with the `if-else` statement.

```
In[8]:  temperature = 68
        if temperature > 100:
            print('Water will boil')
        else:
            print('Water will not boil')
```

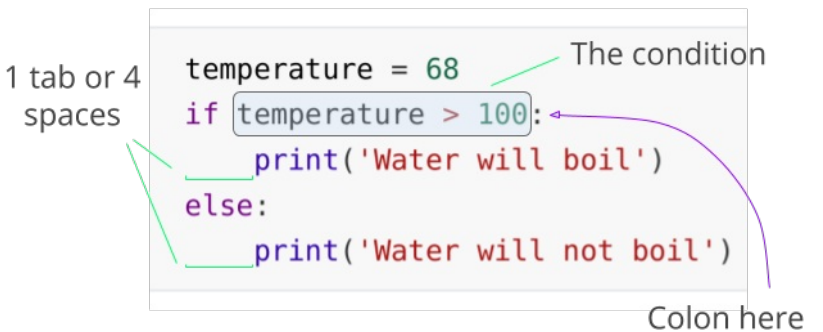
```
Out[8]: 'Water will not boil'
```

- The statement `temperature > 100` is something that is either `True` or `False`.
- If it is `True`, the if-block gets run. If it is `False`, the else-block gets run.

Conditions


Start
Condition
True
False
If block runs
Else block runs
Algorithm continues

Conditions


1 tab or 4 spaces
temperature = 68
if temperature > 100:
print('Water will boil')
else:
print('Water will not boil')
The condition
Colon here

Conditions

- A condition can be anything that is unambiguously `True` or `False`.
- Common conditions include:
  - Are two numbers the same? (`==`)

    ```
    if number1 == number2:
        ...
    ```

  - Is one number greater than (`>`), less than (`<`), greater than or equal to, (`>=`) or less than or equal to (`<=`) another number?

    ```
    if number1 <= number2:
        ...
    ```

Conditions

- Continued...
  - Are two strings the same?

```
if string1 == string2:
    ...
```

  - Is an item found inside a list?

```
if item in my_list:
    ...
```

  - Is one number divisible by another?

```
if (number1 % number2) == 0:
    ...
```

  - ... and more

Conditions

- Once inside an `if`-block, there's nothing to stop you using a second (or third...) `if`-`else` condition

```
In[9]:  temperature = 68

        if temperature > 100:
            print('Water will boil')

        else:
            if temperature < 0:
                print('Water will freeze')
            else:
                print('Water will not boil or freeze')
```

```
Out[9]: 'Water will not boil or freeze'
```

New topic: Loops

Loops

- Say we had test results for multiple students

```
score1 = 68
score2 = 54
score3 = 88
score4 = 36

if score1 > 50:
    print('pass')
else:
    print('fail')

if score2 > 50:
    print('pass')
else:
    print('fail')

...
```

Loops

- There is a better way! First, organise the scores into a list.

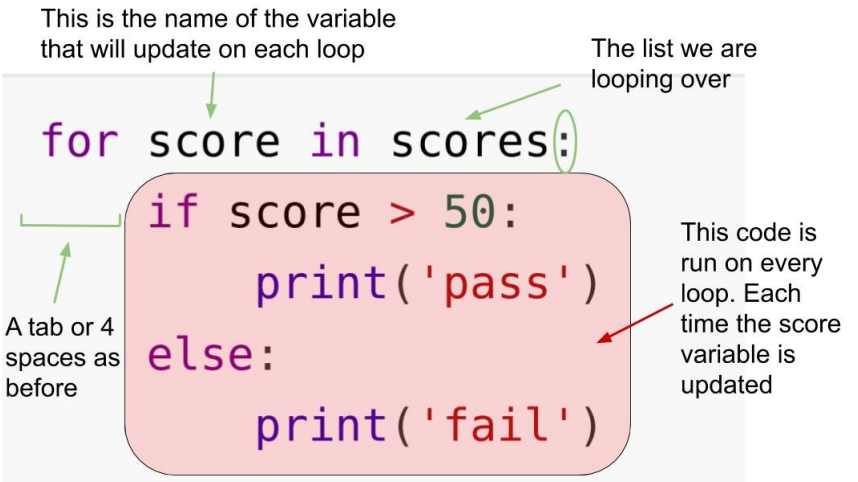
```
scores = [68, 54, 88, 36]
```

- Then perform a *loop* over the list

```
In[10]:  for score in scores:
             if score > 50:
                 print('pass')
             else:
                 print('fail')
```

```
Out[10]: 'pass'
         'pass'
         'pass'
         'fail'
```

For loops


This is the name of the variable that will update on each loop
The list we are looping over
for score in scores:
if score > 50:
print('pass')
else:
print('fail')
A tab or 4 spaces as before
This code is run on every loop. Each time the score variable is updated

For loops

- Loops can be used to go over each item in any list

```
In[11]:  cities = ['Glasgow', 'Edinburgh', 'Aberdeen', 'Dundee']
         for city in cities:
             if city == 'Edinburgh':
                 print(city + '!!')
             else:
                 print(city)
```

```
Out[11]: 'Glasgow'
         'Edinburgh!!'
         'Aberdeen'
         'Dundee'
```

For loops - `range()`

- Another common way to perform loops is using the `range()` function.
- `range(n)` can be thought of as a list containing numbers `[0, 1, ..., n-1]`. Both of the following code snippets produce the same result.
- `for i in range(n):` means `i` will run from `0` to `n`.

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

- Same as this:

```
# snippet 2
for i in [0, 1, 2, 3, 4]:
    print(i)        # also prints 0, 1, 2, 3, 4
```

For loops - `range()`

- By default, `range(n)` starts at zero, and runs to `n-1` in steps of 1.
- However, you can also set a starting number:

```
for j in range(2, 7):
    print(i)        # prints 2, 3, 4, 5, 6
```

- And also a step:

```
for k in range(10, 20, 2):
    print(k)        # prints 10, 12, 14, 16, 18
```

- Setting the start as zero is the same as the normal behaviour:

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

For loops - `range()`

- The `range` function can be useful when you want to access different lists.

```
In[12]: scores = [68, 54, 88, 36]
        names = ['Cathy', 'Azi', 'Samir', 'Ted']

        for i in range(len(names)):

            score = scores[i]
            name = names[i]

            if score > 50:
                out_string = name + ' scored ' + str(score) + ' (pass)'
            else:
                out_string = name + ' scored ' + str(score) + ' (fail)'

            print(out_string)
```

For loops

```
Out[12]: 'Cathy scored 68 (pass)'
         'Azi scored 54 (pass)'
         'Samir scored 88 (pass)'
         'Ted scored 36 (fail)'
```

While loops

- In Python there are actually two types of loop. What you've seen so far are called `for` loops.
- `While` loops are the second kind. Here the code will loop again if some condition is true.
- The loop will only stop when the condition is no longer true.

```
In[13]:  x = 0
         while x <= 5:
             print(x)
             x = x + 1
```

```
Out[13]: 0
         1
         2
         3
         4
```

While loops

- Some variable will have to change on each iteration, to make sure the loop stops at some point.

```
In[14]:  y = 2
         while y <= 100:
             print(y)
             y = 2 * y
```

```
Out[14]: 2
         4
         8
         16
         32
         64
```

Infinite while loops

- If the `while` condition is always `True`, the code could loop forever.
- For example, if the variable is never changed

```
# will print "x still 0 ..." forever
x = 0
while x <= 5:
    print('x still 0 ...')
```

- Or the variable is changed, but the condition is always still true

```
# will print "x still negative ..." forever
x = 0
while x <= 5:
    x = x - 1
    print('x still negative ... ')
```

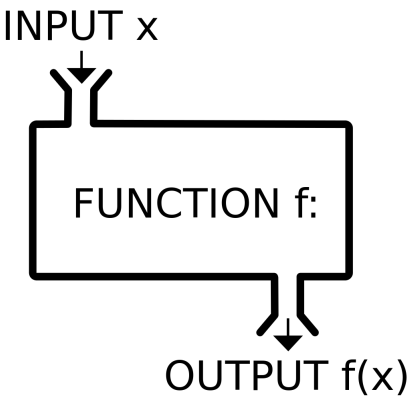
Infinite while loops

- If this happens to you, you can tell Python to stop by 'interrupting the kernel'.

For loops vs while loops

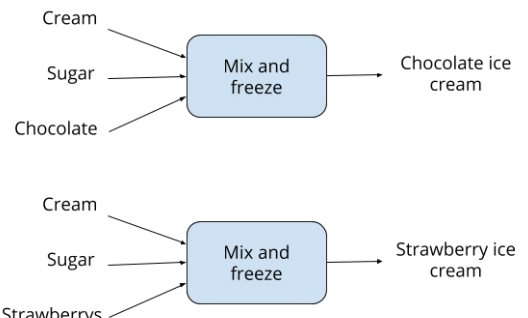
	For loops	While loops
Example usage	`for item in shopping_list:`	`while x <= 5:`
How to identify	Uses `for` keyword.	Uses `while` keyword.
What is does	Loops over each item in a fixed, known sequence.	Loops until some condition is no longer true.
Will it end?	Guaranteed to end.	Not guaranteed to end.

New topic: functions



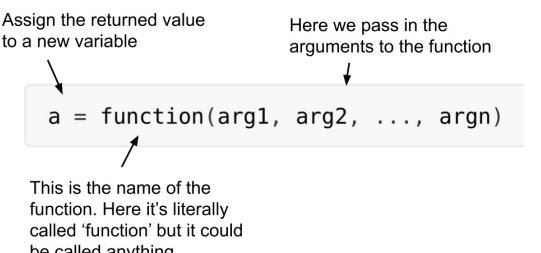
Functions

- Functions are an essential part of most programming languages
- They provide several benefits:
  - Allows you to split up code into simple, reusable blocks.
  - Makes code more readable and neat.
  - Useful in siuations where you want to take in inputs and produce outputs.



Functions

- Functions take inputs and produce outputs.
- In programming jargon, functions take in **arguments** and then **return** something at the end.
- To call a function, we need to type the function name, and then pass in the arguments.
- The returned value can be printed or stored to a new variable.



Functions

- You've have already seen some examples of built-in functions.

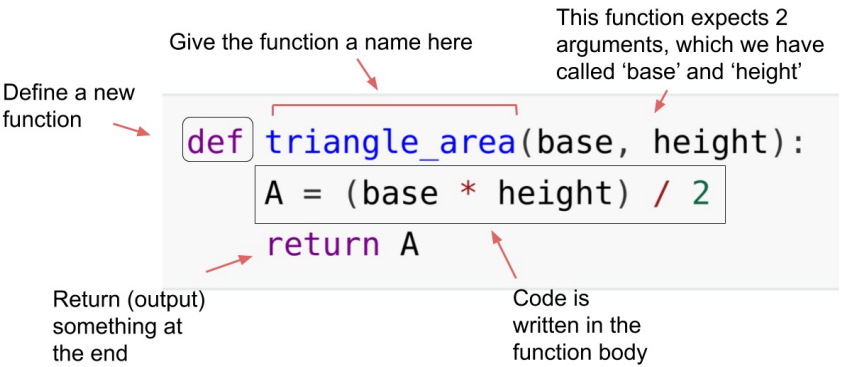
Function	Arguments	Returns
`range()`	A single integer, n	A sequence of numbers from 0 to $n-1$.
`len()`	A single list	The length of that list
`type()`	Any single python object	The type of that object
`print()`	Any number of python objects	Nothing

Functions

- It is also possible to write your own custom functions.
- The code below shows how to define a function called `triangle_area` which takes in the base and height of a triangle, and returns the area of said triangle.

```
def triangle_area(base, height):
    A = (base * height) / 2
    return A
```

Functions


Give the function a name here
This function expects 2 arguments, which we have called ‘base’ and ‘height’
Define a new function
def triangle_area(base, height):
A = (base * height) / 2
return A
Return (output) something at the end
Code is written in the function body

Functions

- Once defined, the new function `triangle_area` can be called anywhere in your code. Either directly on numbers

```
In[15]:  print(triangle_area(12, 10))
```

```
Out[15]: 60
```

- Or on stored variables

```
In[17]:  B = 14
         H = 11
         A = triangle_area(B, H)
         print(A)
```

```
Out[17]: 77
```

Functions

- A temperature in Celsius (C) can be converted into Fahrenheit (F) by using the formula

$$F = 1.8 \times C + 32$$

- ... and backwards

$$C = \frac{F - 32}{1.8}$$

```
def cel_to_fah(C):
    F = 1.8 * C + 32
    return F

def fah_to_cel(F):
    C = (F - 32) / 1.8
    return C
```

Functions

```
In[18]:  def cel_to_fah(C):
             F = 1.8 * C + 32
             return F
```

```
In[20]: temp_in_cel = [-5, 0, 20, 40, 100]
        temp_in_fah = []

        for tc in temp_in_cel:
            tf = cel_to_fah(tc)
            temp_in_fah.append(tf)

        print(temp_in_fah)
```

```
Out[20]: [23.0, 32.0, 68.0, 104.0, 212.0]
```

Functions

- A function doesn't necessarily have to take any inputs.

```
def say_happy_birthday():
    print('Happy birthday')
    return 0
```

- It also doesn't necessarily have to return anything

```
def happy_birthday_to(name)
    birthday_string = 'Happy birthday, ' + name
    print(birthday_string)
```

- By default, if you don't tell python to return anything, a special variable called `None` is returned.

Functions

- You can also specify arguments directly, which allows you to pass them in any order.

```
In[18]:  def av_speed(distance, time):

             if time < 0:
                 print('Invalid time passed')
                 return None

             else:
                 return distance / time

         print(av_speed(time=9.87, distance=100))
```

```
Out[18]: 10.131712259371835
```

- This is known as passing **keyword arguments**.

Functions

- A function can also be defined with *optional* arguments that just resort back to a default if nothing is passed.

```
In[19]:  club = [['Omar',  'Edinburgh'], ['Amy', 'Edinburgh'],
                 ['Chris', 'Edinburgh']]

         def add_new_member(name, location='Edinburgh'):
             club.append([name, location])

         add_new_member('Anita', 'Penicuik')
         add_new_member('Steve')
         print(club)
```

```
Out[20]:  [['Omar',  'Edinburgh'],
           ['Amy',   'Edinburgh'],
           ['Chris', 'Edinburgh'],
           ['Anita', 'Penicuik'],
           ['Steve', 'Edinburgh']]
```

Functions

- Docstrings are used to give some information about what the function is doing

```
def add_new_member(name, location='Edinburgh'):
    '''
    A function to add a new member to the club. Takes
    a name and a location, and adds them to the club record.
    '''
    club.append([name, location])
```

- They can be useful for other people trying to use your code, or yourself in the future.

Thanks!