2. Python Programming Fundamentals

DataKirk

Overview

1. Recap of last time
2. User input
3. Conditions and branching
4. Loops
5. Functions

Recap of last time

- Programming is the art of giving instructions to a computer in order to solve a problem.
- To do this, we write algorithms. An algorithm is a set of very precise instructions that a computer can understand, which help you achieve your goal.
- Computer code must be written in a specific programming language - we will be using Python.
- You can write and run Python on your own computer as well as on our online server. If you want to run Python on your own computer, you will need to download Python.
- When it comes to developing Python code, we recommend using Jupyter. This is an environment where you can write and run small snippets of code.
- However, if you prefer,you can write your code in an IDE. We recommend Visual Studio Code (not Visual Studio, which confusingly, is something different.)

Recap: The print function

- The print function is used to display something on the screen. (it has nothing to do with ink printers!)

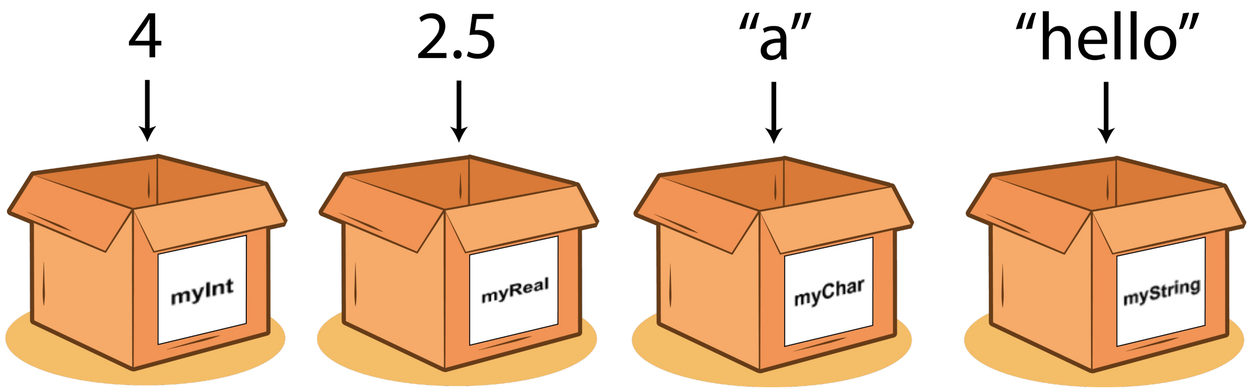
```
print('Hello world!')
```

```
Hello world!
```

- The print function takes in some input and then displays it. The thing that you want to display **must be placed inside a pair of brackets**.

Recap: variables

- Variables are just a named container where you can store a piece of data.



- That data could be a number, some text, or anything else.
- Variables are assigned using the `=` symbol

```
name = 'Anita'
age = 48
print(name, age)
```

```
Anita 48
```

Recap: strings

- A string is just the programming name for a piece of text.

- Whenever you create a string, it must be enclosed in quotation marks.

This is correct!

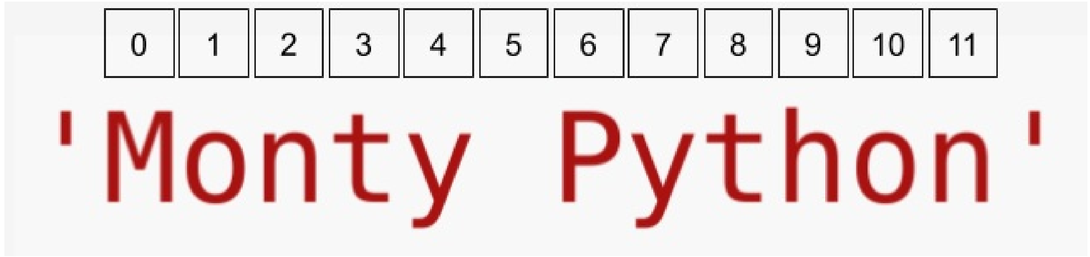
```
my_text = 'hello, nice to meet you'
```

This is incorrect!

```
my_text = hello, nice to meet you
```

Recap: strings

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




```
text = 'Monty Python'
print(text[0])
print(text[6])
print(text[3:7])
```

```
M
P
ty Py
```

Recap: lists

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

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

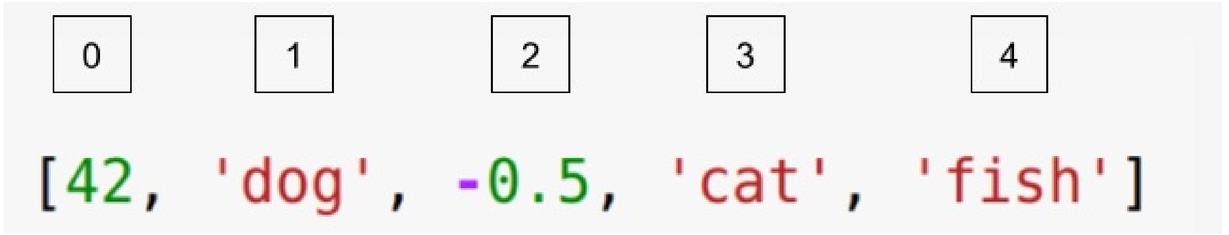
- Lists have a length, which can be found using the `len()` function.

```
list_length = len(my_list)
print(list_length)
```

```
5
```

Recap: lists

- Individual items or sub-lists can be accessed by indexing.




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

```
dog
[-0.5, 'cat']
```

Recap: lists

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

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

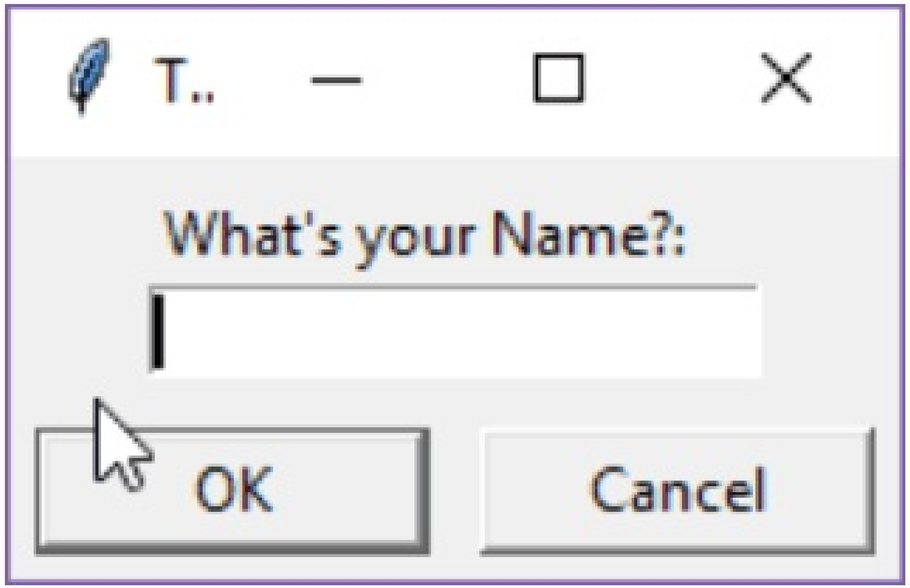
```
['eggs', 'milk', 'apples', 'beans', 'pizza', 'rice']
```

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

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

```
['Kane', 'Sterling', 'Rashford', 'Sancho']
```

New topic: user input



User input

- Sometimes it's useful to gather information from a user when your algorithm runs, for example getting them to type in a password.

- In Python, this can be done with the special function called `intput()`.

```
answer = input("What's your name? ")
print(answer)
```

```
What's your name? Ed
Ed
```

- When your code hits a line with a call to `input()`, the user will be prompted to type something in. You can display a message to be presented to the user at this time by passing it an an argument to the `input('your message here')` function.

User input

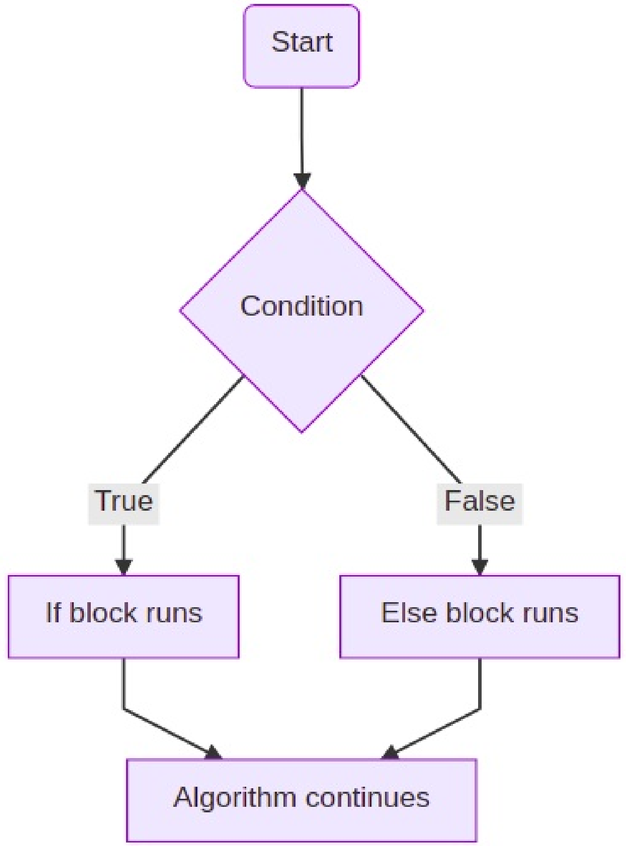
- Note that whatever the user types in will be saved as a *string*. If you want it to be a number, you have to explicitly convert it to a number using `float()`.

```
height_cm = input('What is your height in cm? ')
weight_kg = input('What is your weight in kg? ')

height_cm = float(height_cm)
weight_kg = float(weight_kg)

height_m = height_cm / 100
BMI = weight_kg  / (height_cm ** 2)

print('Your BMI is: ', BMI)
```

```
What is your height in cm? 182
What is your weight in kg? 78
Your BMI is:  23.54788069073783
```

New topic: conditions

Conditions


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

Conditions

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

```
temperature = 68

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

else:
    print('Water will not boil')
```

```
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

1. We begin an `if`-statement with the word `if`

2. This is then followed by the *condition*. This has to be something that is either `True` or `False`

3. The condition is followed by a colon `:`

4. Next, we write the code you want to run if the statement is `True`. All this code **must be indented by 1 tab or 4 spaces**. Indentation is *key* in Python for determining context

5. Then we can optionally use the word `else`, which should be at the same level of indentation as `if`, followed by a colon. After indenting again, all code written here will be executed if the statement is `False`

6. Finally, Python will continue to run any code written at the base level of indentation

What kind of conditions can we build?

Here are some examples of conditions with numbers:

Type	Pattern	Example
Is one number greater than another?	`a > b`	`if my_number > 5:`
Is one number less than another?	`a < b`	`if my_number < 5:`
Is one number greater than or equal to another?	`a >= b`	`if my_number >= 5:`
Is one number less than or equal to another?	`a <= b`	`if my_number <= 5:`
Is one number equal to another?	`a == b`	`if my_number == 5:`

- **WATCH OUT!** `==` asks whether two things are equal, `=` is used for assigning a variable.

Conditions with strings

- Numbers aren't the only thing that can be used to create a true/false statement: you could also use strings.

- When considering two strings, the `==` operator asks whether they are identical.

```
password = input('Please type in your password: ')

if password == 'qwerty':
    print('Password Accepted')

else:
    print('Sorry, that is the wrong password.')
```

```
Please type in your password: 12345678
Sorry, that is the wrong password.
```

Conditions with lists

- We could also create a condition from a list.

- One common list-based condition is to ask whether an item is contained `in` a list.

```
items = ['sword', 'health potion', 'shield', 'magic spell']

print('A giant snake appears! :O')

if 'sword' in items:
    print('Begin your battle!')

else:
    print('Oh no! You got eaten')
```

```
A giant snake appears! :O
Begin your battle!
```

Nested conditions

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

```
temperature = 68

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

else:

    if temperature < 0:
        print('Water will freeze')

    else:
        print('Water will neither boil nor freeze')
```

```
Water will neither boil nor freeze
```

Combining conditions: `or` and `and`

- You can also combine multiple conditions together with the words `or` and `and`

```
answer1 = input("What is your mother's maiden name? ")
answer2 = input("What is the name of your first pet? ")

if answer1 == 'Smith' and answer2 == 'Spike':
    print('Security questions passed')

else:
    print('Security questions failed')
```

```
What is your mother's maiden name? Smith
What is the name of your first pet? Spike
Security questions passed
```

New topic: Loops

Loops

- Often in programming we want to perform the same task on many items.

```
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

- This soon becomes impractical! We don't want to write out the same code many times. Instead, we would be better off "looping" over all the scores.

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

for score in scores:

    if score > 50:
        print('Pass')
    else:
        print('Fail')
```

```
Pass
Pass
Pass
Fail
```

Loops

```
for score in scores:
```

- `scores` is the list of test scores that we created.
- `score` is our looping variable - it can be called anything you like.
- On each loop cycle, the looping variable gets updated to be the next item in the specified list.

Loops

- The code we specify underneath the loop declaration then gets executed again and again. And on each loop, the variable `score` is changing.

```
if score > 50:
    print('Pass')
else:
    print('Fail')
```

- This code must be indented by a tab or four spaces.

`range()`

- Often you will find yourself in a situation where you want to perform a loop N times.

- Here the `range()` function can be very useful. You can think of `range(N)` as a list of numbers going from 0 to $N-1$.

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

range()

- This can be useful, for example, when your want to use a variable to index two or more lists.

```
scores = [68, 54, 88, 36]
names = ['Cathy', 'Azi', 'Samir', 'Ted']
n_students = len(names)

for i in range(n_students):

    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)
```

range()

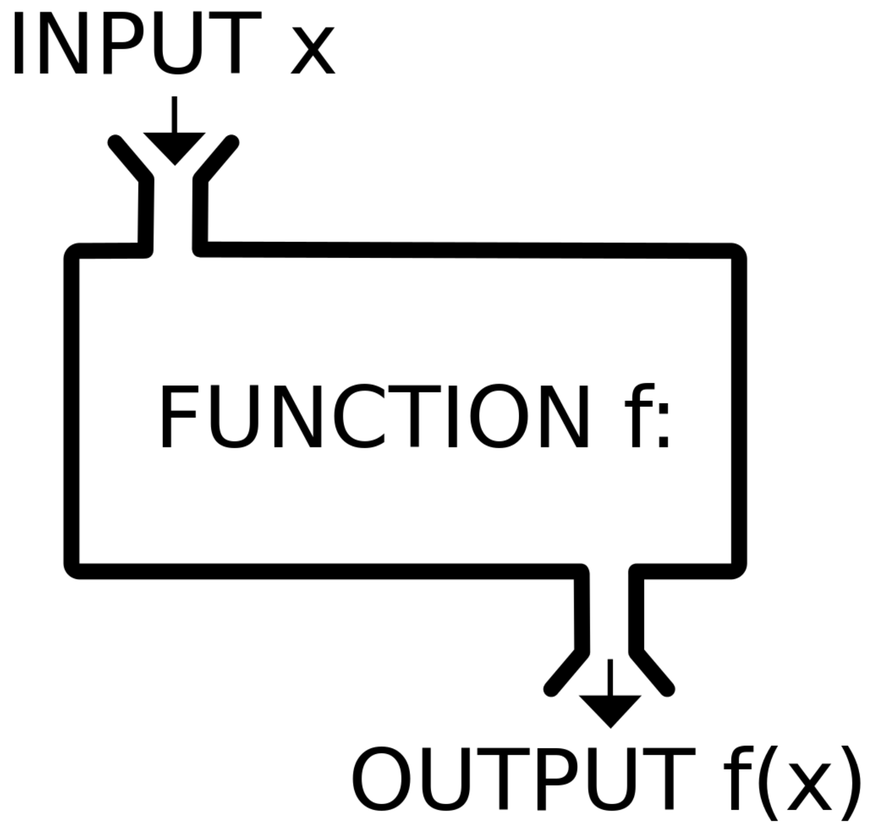
- Here's an example usuage of `range()` to produce the fibonnacci numbers:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

```
numbers = [1, 1]

for i in range(10):
    new_number = numbers[i] + numbers[i + 1]
    numbers.append(new_number)
    print(numbers)
```

```
[1, 1, 2]
[1, 1, 2, 3]
[1, 1, 2, 3, 5]
[1, 1, 2, 3, 5, 8]
[1, 1, 2, 3, 5, 8, 13]
[1, 1, 2, 3, 5, 8, 13, 21]
[1, 1, 2, 3, 5, 8, 13, 21, 34]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

New topic: Functions

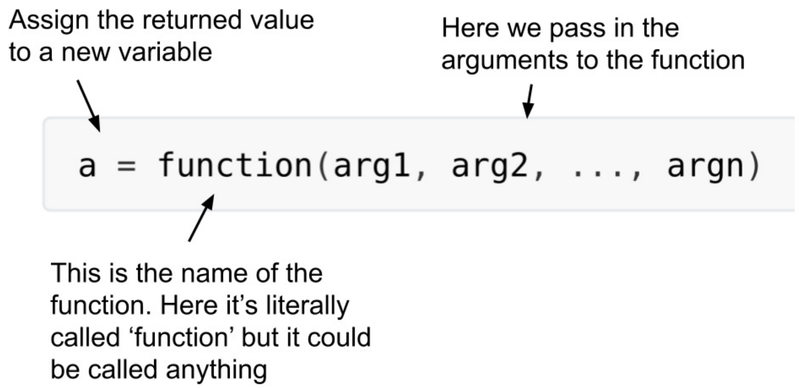


Functions

- Functions are an essential part of all programming languages. A function is like a template for performing some operation, which can be used as many times as you like.

- Functions are like a little reusable unit of code that take in some input(s), perfrom some computation, and then produce some output(s). In programming jargon function inputs are called *arguments*, and we say that a function *returns* the output.

- In Python, you can call a function like this



Functions

- The good news is, you've already seen quite a few functions!

Function	Input	What it does	Output
`print()`	Any python object	Displays that object to the screen	Nothing
`len()`	A list or string	Calculates the length of the list/string	A number
`float()`	A string	Converts the string to a number	A number
`input()`	A string	Displays the message and prompts the user for input	The string typed in by the user
`range()`	A whole number	Produces a sequence of numbers from 0 to N-1	A list-like object
`sum()`	A list of numbers	Adds all the numbers in the list	The sum of the numbers

Functions

- These are all examples of *built-in* functions - they exist already in Python for us to use. But the real power of functions is that you are able to *define your own*! Take a look at this function. It joins two strings together with a space in between them.

```
def join_strings(string_a, string_b):

    joined_string = string_a + " " + string_b

    return joined_string
```

- This function expects two arguments, which should be atrings. It will then take those two strings and concatenate them together with a space in between.

- This result is saved to a variable and returned.

Functions

- Once we have defined a function, we can then use it anywhere else in our code.

```
output1 = join_strings('The', 'DataKirk')
output2 = join_strings('Computer', 'Science')
output3 = join_strings('Pulp', 'fiction')

print(output1)
print(output2)
print(output3)
```

```
The Datakirk
Computer science
Pulp Fiction
```

Functions - more examples

- Here's an example of a function that takes in the price of an item, and the amount that a customer has paid, and calculates the change. It also checks whether the cutsomer has paid enough, and if not returns a message asking for more money.

```
def calculate_change(price, paid):

    if paid >= price:
        return paid - price
    else:
        return 'Please pay more'

print(calculate_change(1.99, 2.00))
print(calculate_change(14.95, 10.00))
```

```
0.01
Please pay more
```

Functions - more examples

- Take a look at this function which calculates the average speed given a distance travelled and a time taken. Can you see how it works? What will be produced by the two print statements at the end?

```
def average_speed(distance, time):

    if time < 0:
        return 'Invalid time!'

    if distance < 0:
        return 'Invalid distance!'

    else:
        return distance / time

print(average_speed(150, 10))
print(average_Speed(10, -5))
```

Functions - more examples

- Here are some more exmples of functions we could define

```
def list_average(numbers):
    list_sum = sum(numbers)
    list_len = len(numbers)
    average = list_sum / list_len
    return average

print( list_average( [1, 1, 1] ) )
print( list_average( [1, 2, 3, 4, 5] ) )
print( list_average( [0, 500, 1000]  ) )
```

```
1
3
500
```

Functions - a longer example

```
def fibonacci_numbers(N):

    if N < 0:
        return 'Invalid input'

    if N == 0:
        return []

    if N == 1:
        return [1]

    numbers = [1, 1]

    for i in range(N - 2):
        new_number = numbers[i] + numbers[i + 1]
        numbers.append(new_number)

    return numbers
```

Functions - more examples

```
output1 = fibonacci_numbers(5)
output2 = fibonacci_numbers(10)
output3 = fibonacci_numbers(-1)

print(output1)
print(output2)
print(output3)
```

```
[1, 1, 2, 3, 5]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584]
Invalid input
```

Thank You