cct College Dublin
Computing • IT • Business

PROGRAMMING: OBJECT-ORIENTED APPROACH 

VARIABLES AND DATA TYPES

- Press `Space` to navigate through the slides
- Use `Shift+Space` to go back
- Save as **PDF**:
  - Open **Chrome** then **click here**
  - Press **Ctrl+P**/**Cmd+P** to print
    - Destination: **Save as PDF**
    - Layout: **Landscape**
  - Press **Save** button

VARIABLES AND DATA TYPES

- When you are writing code you will often need to take data and use it/manipulate it in different ways to give a meaningful output.
- **Variables** and **Data Types** are key to understanding python (and any programming language).

VARIABLES

- If you remember from algebra – variables are just some characters that were used to represent numbers.
- For example `x = 5` and then you could use it to say `5 + x = 10`.
- Variables in Python work much the same way, except you can **store data** (sometimes this is referred to as **alias**) of different **types** (text, numbers, decimals etc.)

CREATING AND UPDATING VARIABLES

- The basic syntax for creating variables looks like this:

```
variable_name = (some data)
```

- You put the *variable name* **on the left** (what you type in when you want to get the data)
- The *data* to store/alias **on the right** with a **single** '=' in between.
- If you wanted to create a name variable and store someone's name to print out later you could do this:

```
name = "Kieran" # Created a variable called name
print(name) # Prints: Kieran
```

CREATING AND UPDATING VARIABLES

- You can also go in and update a value later on by assigning it some new data:

```
name = "Kieran" # Create/instantiate name variable
print(name) # Prints: Kieran

name = "Bob" # Reassign name variable to 'Bob'
print(name) # Prints: Bob
```

VARIABLE NAMES

- Naming variables can be hard sometimes, here are the general rules on what you can and cannot do with them.
- **Can include:**
  - All upper and lowercase letters
  - Underscores
  - Numbers (But not as the first character): 1 2 3 4 5 6 7 8 9 0
- **Cannot include:**
  - Dots (Possible but means something different in python): .
  - Reserved Characters (Characters that already do something in python): + | & * $ # @ ( ) ? < > = ' " \ / ^ ! ~ _
  - The first character as a number

VARIABLE NAMES

MAKE VARIABLE NAMES USEFUL:

- Constantly reading `x`, `j`, `i`, `k` and other single letter variables, they all start to meld together.
- You can easily be confused because they give you no indication of what the variable actually represents (usually).

VARIABLE NAME GUIDELINES

- Here are some guidelines to help create better variable names:
- Use the **4 W's** (Who, What, When, Where)

  - **Who:** If your variable represents someone or something then use their name i.e.

```
p = "Lincoln" # Bad, what does p even mean in this context?

president = "Lincoln" # Now you know what I am talking about without seeing the
```

  - **What:** what the variable is in this context.

```
dx = 5 # If you know the notation this might make sense but what if someone doe

delta_x = 5 # You know exactly what the variable represents
```

VARIABLE NAME GUIDELINES

- Use the **4 W's** (Who, What, When, Where)

  - **When:** this may be less apparent right now but when we look at loops later this naming convention can be useful.

```
d = "17-10-2019" # We can infer it's a date, but what date is it?

current_date = "17-10-2019" # Now we know it represents the current date
```

  - **Where:** Only useful in specific use cases but still better than nothing.

```
l = (51.0447, 114.0719) # ¯\_(ツ)_/¯ Who knows what this variable represen

user_coordinates = (51.0447, 114.0719) # Ahh it's user coordinates
```

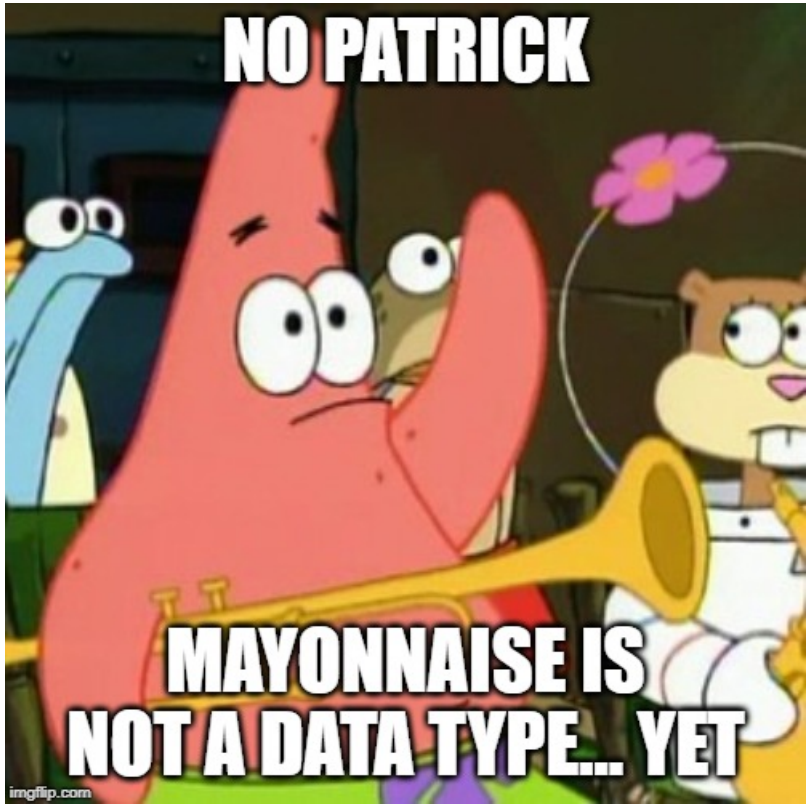
VARIABLE NAME GUIDELINES

- This can sometimes be difficult but if you do it then others looking at your code will hate you much less when your code breaks.

source

DATA TYPES

- Python can store many different ***data types***, we have already seen a few in our examples.
- As you saw we can store basic (primitive) ***data types***, such as:
  - text – **string(s)** or **str**
  - whole numbers – **integers** or **ints**
  - decimals – **float(s)**
  - or ***collections*** of data types
- Later on we will learn how to create your own data types.



DATA TYPES

- If you are ever unsure you can actually see the 'type' of a variable by using the type() function. For example:

```
variable_1 = 5 # An integer or int
variable_2 = "hello" # A string or str

print(type(variable_1)) # Prints <class 'int'>
print(type(variable_2)) # Prints <class 'str'>
```

PRIMITIVE DATA TYPES

- **Integer (or int)**
  - Any positive or negative **whole number**:

```
number_1 = 1 # Positive int

number_2 = -2 # Negative int

number_3 = 123665568654756475647465745 7 # Large positive i

number_4 = -432587965423943857612347861 # Large negative i
```

- **Float**
  - Any positive or negative **decimal number**

```
number_1 = 1.5 # Positive float

number_2 = -2.345 # Negative float

number_3 = 12366556.7893 # Large positive float

number_4 = -432587965423.3457 # Large negative float
```

PRIMITIVE DATA TYPES

- **String**
  - Text; Note that this can include numbers

```
variable_1 = "This is a string" # Anything inside the "" is part of the st

variable_2 = 'This is a string' # You can also use '' to create strings
```

- **Boolean**
  - Used to indicate **True or False**; Note that True and False also correspond to 1 and 0 respectively

```
# Booleans are created by just writing true or false
# NOTE: Capitalise the first letters!

variable_1 = True # True or 1

variable_2 = False # False or 0
```

COLLECTIONS

- **Collections** are data types that allow you to store multiple variables (referred to as elements) inside of them.
- This is convenient in many cases to store data that is logically grouped together like a shopping list, or names of people in a group/class.
- I will mention **3 of the most common collections** but there are actually **many** more available in python to cover a wide variety of use cases.

COLLECTIONS: LISTS

- **Lists**
  - Allow you to **store** and **change** values (Sometimes called mutating values) that are added to it.
  - Here is an example of setting up various types of lists

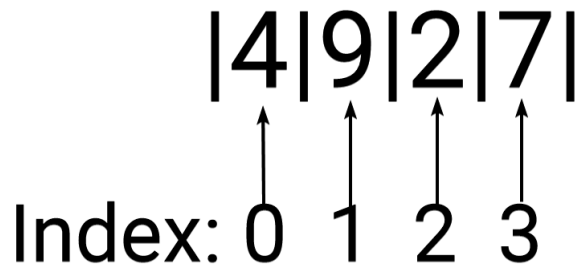
```
variable_1 = []                # An empty list

variable_2 = [2, 4, 6, 8]      # A list of ints

variable_3 = [2, "two", 2.1] # A list of mixed data types

# You can use the list.append() method to add elements to an existing list
variable_2.append(10) # variable_2 is now: [2, 4, 6, 8, 10]
```

  - You can access values stored in a list by using their ***index***.
  - Indices are counted **from zero up** as you add elements to the list.

COLLECTIONS: LISTS

- **Lists**
  - Take for example the following list:

```
variable_4 = [4, 9, 2, 7] # This is the list used in the next ima
```



  - So to access the *elements* you would use the following format:

```
print(variable_4[0]) # Prints: 4

print(variable_4[1]) # Prints: 9

print(variable_4[2]) # Prints: 2

print(variable_4[3]) # Prints: 7
```

COLLECTIONS: TUPES

- **Tuples**

  - Tuples are similar to lists, the biggest difference being that they are *immutable*, me *elements* cannot be updated after they have been added.

  - Also *Elements* **cannot** be added to tuples after they have been created. Tuples have syntax to list for creating them, and the exact same syntax for accessing elements:

```
variable_1 = ()                # An empty tuple
variable_2 = (2, 4, 6, 8)      # A tuple of ints
variable_3 = (2, "two", 2.1) # A tuple of mixed data types
variable_4 = (4, 9, 2, 7)      # This is the same as the list used in the previou

# Accessing and printing values
print(variable_4[0]) # Prints: 4
print(variable_4[1]) # Prints: 9
print(variable_4[2]) # Prints: 2
print(variable_4[3]) # Prints: 7
```

COLLECTIONS: DICTIONARIES

- **Dictionaries**
  - Dictionaries are what's called a key-value store data structure.
  - Dictionaries are also *mutable* like lists, which means you can add and update *elements* as you please.
  - What this means is that instead of using indices that go up every time something is added, they use **key's** that correspond to **values** to access & insert data

COLLECTIONS: DICTIONARIES

Key ⟶ Value

- Must be a string
- Used to access corresponding value

- Can be any type
- Accessed through the corresponding value

Example

"name" ⟶ "John Doe"

```
variable_1 = {} # Empty dictionary

variable_2 = {"name": "John Doe"} # Assigning the key 'name' to the value 'John Do

# Dictionaries can contain values of different types, but keys must be strings
variable_3 = {"name":"John Doe", age: 21, "net worth": 5213.4}

# To access a value, use the key as you would an index
print(variable_3["name"]) # Prints: John Doe

# Adding new key-value pairs to a dictionary uses the same syntax
variable_2["age"] = 21 # variable_2 is now: {"name":"John Doe", age: 21}
```

MUTABILITY

- Mutability, or the ability to mutate/change an element once it has been added to a collection is an important distinction that can cause many common errors.
- Lists are a *mutable* data structure, meaning their *elements* can be updated while they are part of the list.
- This means that they should **only** be used in cases where this makes sense, for example a list of configuration information.
- Tuples on the other hand are immutable, meaning once an *element* is in a tuple it will stay as it is, this is useful for places where data shouldn't be changing.
  - For example if you wanted to store a list of Dates of birth, you wouldn't want someone accidentally updating them if they thought it was a list of dates for something else and so a tuple would likely be more appropriate.

TYPE CASTING

- In python you can convert data between data types.
- Python is what's called a **strongly typed** language, what this means is that python won't do any converting unless you **explicitly** ask it to. For example

```
variable_1 = "4" # Currently is the string '4'

2 + variable_1 # Would throw an error

print(2 + int(variable_1)) # Would convert the string 4 to an int and then print
```

EXERCISE TIME

- Check out the exercises.py for some simple exercises to try out.