

Introduction to python

Part 2

Recap

- Variables
- Types
- Arithmetic operators
- Boolean logic
- Strings
- Printing
- Exercises

Structure of Part 2

- Part 2.1: Lists
- Part 2.2: Dictionaries
- Part 2.3: Sets
- Part 2.4: If Else
- Part 2.5: Loops
- Part 2.6: Functions
- Part 2.7: Classes
- Part 2.8: Exercises

Lists

- One of the most useful concepts
- Group multiple variables together (a kind of **container!**)

```
fruits = ["apple", "orange", "tomato", "banana"] # a list of strings
print(type(fruits))
print(fruits)
```

```
<class 'list'>
['apple', 'orange', 'tomato', 'banana']
```

Indexing a list

- Indexing – accessing items within a data structure

```
fruits[2]
```

```
'tomato'
```

- Indexing a list is not very intuitive...
- The first element of a list has an **index 0**

Index:	0	1	2	3
List:	apple	orange	tomato	banana

Quick quiz

What will **fruits[3]** return?

```
fruits = ["apple", "orange", "tomato", "banana"] # a list of strings
print(type(fruits))
print(fruits)
```

```
<class 'list'>
['apple', 'orange', 'tomato', 'banana']
```

Quick quiz

What will this return?

```
fruits[4]
```


IndexError

Traceback (most recent call l

ast)

<ipython-input-14-b8c91da6ba3a> in <module>()

----> 1 fruits[4]

IndexError: list index out of range

Data structure sizes

Make sure you are always aware of the sizes of each variable!

This can easily be done using the **len()** function.

It returns the length/size of any data structure

```
len(fruits)
```

4

Is a tomato really a fruit?

```
fruits[2] = "apricot"  
print(fruits)
```

```
['apple', 'orange', 'apricot', 'banana']
```

Furthermore, we can modify lists in various ways

```
fruits.append("lime")    # add new item to list  
print(fruits)  
fruits.remove("orange")  # remove orange from list  
print(fruits)
```

```
['apple', 'orange', 'apricot', 'banana', 'lime']  
['apple', 'apricot', 'banana', 'lime']
```

Lists with integers

range() - a function that generates a sequence of numbers as a list

```
nums = list(range(10))  
print(nums)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
nums = list(range(0, 100, 5))  
print(nums)
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85,  
90, 95]
```

Slicing lists

- Slicing – obtain a particular set of sub-elements from a data structure.
- Very useful and flexible.

```
print(nums)
print(nums[1:5:2]) # Get from item 1(starting point) through item 5(end point, not included) with step size 2
print(nums[0:3]) # Get items 0 through 3(not included)
print(nums[4:]) # Get items 4 onwards
print(nums[-1]) # Get the last item
print(nums[::-1]) # Get the whole list backwards
```

```
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
[5, 15]
[0, 5, 10]
[20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
95
[95, 90, 85, 80, 75, 70, 65, 60, 55, 50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0]
```

Lists – helpful functions

- Makes them extremely useful and versatile

```
print(len(nums))    # number of items within the list
print(max(nums))    # the maximum value within the list
print(min(nums))    # the minimum value within the list
```

```
20
95
0
```

Lists can be of different types

- Not very useful, but possible

```
mixed = [3, "Two", True, None]  
print(mixed)
```

```
[3, 'Two', True, None]
```

Mutability

Mutable object – can be changed after creation.

Immutable object - can **NOT** be changed after creation.

Quick quiz

- Are lists mutable?

Tuples

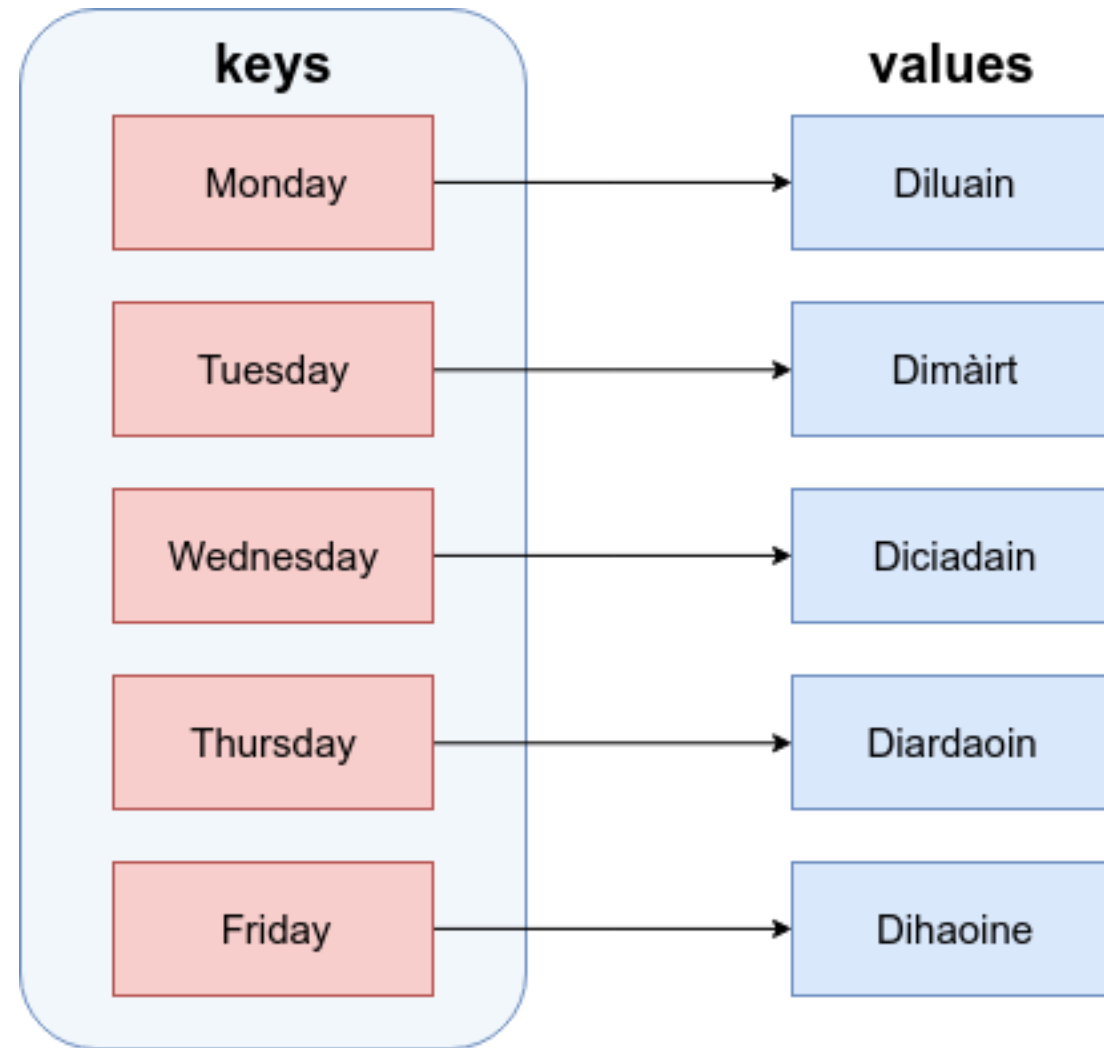
- Effectively lists that are immutable (i.e. can't be changed)

```
fruits = ("apple", "orange", "tomato", "banana") # now the tomato is a fruit forever  
print(type(fruits))  
print(fruits)
```

```
<class 'tuple'>  
('apple', 'orange', 'tomato', 'banana')
```


Dictionaries

- Similar to actual dictionaries
- They are effectively 2 lists combined – keys and values
- We use the keys to access the values instead of indexing them like a list
- Each value is mapped to a unique key



Dictionary definition

Defined as comma separated **key : value** pairs:

```
mydict = {key1: val1,  
          key2: val2,  
          key3: val3}
```

Curly brackets

Comma separated

Dictionary properties

- Values are mapped to a key
- Values are accessed by their key
- Key are unique and are immutable
- Values cannot exist without a key

Dictionaries

Let us define the one from the previous image

```
days = {"Monday": "Diluain", "Tuesday": "Dimàirt",  
        "Wednesday": "Diciadain", "Thursday": "Diardaoin",  
        "Friday": "Dihaoine"}  
print(type(days))  
print(days)
```

```
<class 'dict'>  
{'Monday': 'Diluain', 'Tuesday': 'Dimàirt', 'Wednesday': 'Diciadain',  
 'Thursday': 'Diardaoin', 'Friday': 'Dihaoine'}
```

Accessing a dictionary

Values are accessed by their keys (just like a dictionary)

```
days["Friday"]
```

```
'Dihaoine'
```

Note that they can't be indexed like a list

Altering a dictionary

Can be done via the dictionary methods

```
days.update({"Saturday": "Disathairne"})  
print(days)  
days.pop("Monday")  # Remove Monday because nobody likes it  
print(days)
```

```
{'Monday': 'Diluain', 'Tuesday': 'Dimàirt', 'Wednesday': 'Diciadain',  
'Thursday': 'Diardaoin', 'Friday': 'Dihaoine', 'Saturday': 'Disathairn  
e'}
```

```
{'Tuesday': 'Dimàirt', 'Wednesday': 'Diciadain', 'Thursday': 'Diardaoi  
n', 'Friday': 'Dihaoine', 'Saturday': 'Disathairne'}
```

Keys and Values

It is possible to obtain only the keys or values of a dictionary.

```
print(days.keys())    # get only the keys of the dictionary  
print(days.values()) # get only the values of the dictionary
```

```
dict_keys(['Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'])  
dict_values(['Dimàirt', 'Diciadain', 'Diardaoin', 'Dihaoine', 'Disathairne'])
```

This is useful for iteration.

Sets

- Effectively lists that can't contain duplicate items
- Similar functionality to lists
- Can't be indexed or sliced
- Can be created with `{}` or you can convert a list to a set

```
x = set([1, 2, 3]) # a set created from a list
print(type(x))
print(x)
y = {1, 2, 3}      # a set created directly

x == y             # x and y are the same object
```

```
<class 'set'>
{1, 2, 3}
True
```


If Else

- Fundamental building block of software

```
if answer:
    close program
else:
    continue running program
```

Conditional statement

Executed if answer is True

Executed if answer is False

If Else example

Try running the example below.
What do you get?

```
x = True
if x:
    print("Executing if")
else:
    print("Executing else")
print("Prints regardless of the outcome of the if-else block")
```

Executing if

Prints regardless of the outcome of the if-else block

Indentation matters!

- Code is grouped by its indentation
- Indentation is the number of whitespace or tab characters before the code.
- If you put code in the wrong block then you will get unexpected behaviour

```
x = 10
if x%2 == 0:
    print(x, 'is even!')
    if x%5 == 0:
        print(x, 'is divisible by 5!')
        print('Output only when x is divisible by both 2 and 5.')
    else:
        print(x, 'is not divisible by 5!')
        print('Output only when x is divisible by 2 but not divisible by 5.')
else:
    print(x, 'is odd!')
print('No indentation. Output in all cases.')
```

```
10 is even!
10 is divisible by 5!
Output only when x is divisible by both 2 and 5.
No indentation. Output in all cases.
```

Extending if-else blocks

- We can add infinitely more if statements using **elif**

```
if condition1:  
    condition 1 was True  
elif condition2:  
    condition 2 was True  
else:  
    neither condition 1 or condition 2 were True
```

- elif = else + if which means that the previous statements must be false for the current one to evaluate to true

Bitcoin broker example

```
purchasePrice = float(input("Price at which you have purchased bitcoins: "))
currentPrice = float(input("Current price of the bitcoins: "))

if currentPrice < purchasePrice*0.9:
    print("Not a good idea to sell your bitcoins now.")
    print("You will lose", purchasePrice - currentPrice, "£ per bitcoin.")
elif currentPrice > purchasePrice*1.2:
    print("You will make", currentPrice - purchasePrice, "£ per bitcoin.")
else:
    print("Not worth selling right now.")
```

Quick quiz

- What would happen if both conditions are True?

```
purchasePrice = float(input("Price at which you have purchased bitcoins: "))
currentPrice = float(input("Current price of the bitcoins: "))

if (currentPrice > purchasePrice*0.9):
    print("Not a good idea to sell your bitcoins now.")
    print("You will lose", purchasePrice - currentPrice, "£ per bitcoin.")
elif (currentPrice > purchasePrice*1.2):
    print("You will make", currentPrice - purchasePrice, "£ per bitcoin.")
else:
    print("Not worth selling right now.")
```

For loop

- Allows us to iterate over a set amount of variables within a data structure. During that we can manipulate each item however we want

```
for item in itemList:  
    do something to item
```

- Again, indentation is important here!

Example

- Say we want to go over a list and print each item along with its index

```
fruits = ["apple", "orange", "tomato", "banana"]  
print("The fruit", fruits[0], "has index", fruits.index(fruits[0]))  
print("The fruit", fruits[1], "has index", fruits.index(fruits[1]))  
print("The fruit", fruits[2], "has index", fruits.index(fruits[2]))  
print("The fruit", fruits[3], "has index", fruits.index(fruits[3]))
```

The fruit apple has index 0

The fruit orange has index 1

The fruit tomato has index 2

The fruit banana has index 3

- What if we have much more than 4 items in the list, say, 1000?

For example

- Now with a for loop

```
fruitList = ["apple", "orange", "tomato", "banana"]  
for fruit in fruitList:  
    print("The fruit", fruit, "has index", fruitList.index(fruit))
```

```
The fruit apple has index 0  
The fruit orange has index 1  
The fruit tomato has index 2  
The fruit banana has index 3
```

- Saves us writing more lines
- Doesn't limit us in term of size

Numerical for loop

```
numbers = list(range(10))  
for num in numbers:  
    squared = num ** 2  
    print(num, "squared is", squared)
```

```
0 squared is 0  
1 squared is 1  
2 squared is 4  
3 squared is 9  
4 squared is 16  
5 squared is 25  
6 squared is 36  
7 squared is 49  
8 squared is 64  
9 squared is 81
```

While loop

- Another useful loop. Similar to the for loop.
- A while loop doesn't run for a predefined number of iterations, like a for loop. Instead, it stops as soon as a given condition becomes true/false.

```
n = 0
while n < 5:
    print("Executing while loop")
    n = n + 1

print("Finished while loop")
```

```
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Finished while loop
```

Break statement

- Allows us to go(break) out of a loop preliminary.
- Adds a bit of controllability to a while loop.
- Usually used with an if.
- Can also be used in a for loop.

Quick quiz

How many times are we going to execute the while loop?

```
n = 0
while True: # execute indefinitely
    print("Executing while loop")

    if n == 5: # stop loop if n is 5
        break

    n = n + 1

print("Finished while loop")
```

```
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Executing while loop
Finished while loop
```

Functions

- Allow us to package functionality in a nice and readable way
- reuse it without writing it again
- Make code modular and readable
- Rule of thumb - if you are planning on using very similar code more than once, it may be worthwhile writing it as a reusable function.

Function declaration

keyword

Any number of arguments

```
def functionName(argument1, argument2, argument3, ... argumentN):  
    statements..  
    ..  
    ..  
  
    return returnValue
```

[Optional] Exits the function and returns some value

- Functions accept arguments and execute a piece of code
- Often they also return values (the result of their code)

Function example

```
def printNum(num):  
    print("My favourite number is", num)  
  
printNum(7)  
printNum(14)  
printNum(2)
```

My favourite number is 7
My favourite number is 14
My favourite number is 2

Function example 2

We want to make a program that rounds numbers up or down.

Try to pack the following into a function.

```
x = 3.4
remainder = x % 1
if remainder < 0.5:
    print("Number rounded down")
    x = x - remainder
else:
    print("Number rounded up")
    x = x + (1 - remainder)

print("Final answer is", x)
```

```
Number rounded down
Final answer is 3.0
```

Function example 2

```
def roundNum(num):  
    remainder = num % 1  
    if remainder < 0.5:  
        return num - remainder  
    else:  
        return num + (1 - remainder)  
  
# Will it work?  
x = roundNum(3.4)  
print (x)  
  
y = roundNum(7.7)  
print(y)  
  
z = roundNum(9.2)  
print(z)
```

3.0
8.0
9.0

Function example 3

$$(val - src[0]) \times \frac{dst[1] - dst[0]}{src[1] - src[0]} - dst[0]$$

```
# Generic scale function
# Scales from src range to dst range
def scale(val, src, dst=(-1,1)):
    return (int(val - src[0]) / (src[1] - src[0])) * (dst[1] - dst[0]) + dst[0]

print(scale(49, (-100,100), (-50,50)))
print(scale(49, (-100,100)))
```

24.5

0.49

Python built-in functions

Built-in Functions				
<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

To find out how they work: <https://docs.python.org/3.3/library/functions.html>

Classes

- Important for programming
- Useful, but more advanced
- Will not be taught here due to time limitations .. but there are explanations and examples in the notebooks

Further reading

- LinkedIn Learning:
- Python Essentials Training – more detailed; good if you want to use your own environment – <https://www.linkedin.com/learning/python-essential-training-2/welcome?u=50251009&auth=true>
- Python for Data Science – continues this course; taught with Jupyter Notebooks as well – <https://www.linkedin.com/learning/python-for-data-science-essential-training-part-1/data-science-life-hacks?u=50251009&auth=true>
- Python Crash Course book – more detailed; more exercises
- Python Data Analysis – O'Reilly press
- PEP 8 style: <https://www.python.org/dev/peps/pep-0008/>

Exercise time

- Cool exercises that can test your knowledge(notebooks 2 and 3)
 - There is checking code which checks if you have completed your task. Please don't modify it.
 - You can also do your exercises in your own time at home (Noteable is accessible from everywhere)
-oor you can also ask any questions
- Google is your best friend when coding
 - Extra notebook available with advanced exercises

