# PYTHON CRASH COURSE

SUMMARY

## Table of Contents



## Data types

| | |
|---|---|
| **String** | Ordered sequence of characters between ‘’ or “” e.g. ‘Dancer’, ‘Apples’ |
| **Float** | Numbers with decimal point e.g. 4.6, 6.889 |
| **Integer** | Whole numbers e.g. 3, 6 |
| **List** | Ordered sequence of objects e.g. [10, ‘Roy’, False] |
| **Tuple** | Ordered immutable sequence of objects e.g. (10, ‘Roy’, False) |
| **Set** | Unordered collection of unique objects {‘Roy’, ‘Dolgas’} |
| **Dictionary** | Unordered key-value pair {‘key’:‘value’, key’:‘value’} |
| **Boolean** | Logical values either *True* or *False* |

# Strings

| | |
|---|---|
| **s.upper()** | Changes the case of all characters in the string s to uppercase |
| **s.lower()** | Changes the case of all characters in the string s to lowercase |
| **s.title()** | Changes the case of the string s such that every word starts with a capital case |
| **s.rstrip()** | Removes white spaces from the right side of string s |
| **s.lstrip()** | Removes white spaces from the left side of string s |
| **s.strip()** | Removes white spaces from both left and right sides of string s |
| **s.count(w)** | Returns the count of appearances of string w in the string s |
| **s.split()** | Splits string s into a list of words |
| **S[index1:index2]** | Returns a substring of s starting from the character at index1 and ending at index2-1 |
| **S[index1:]** | Returns a substring of s starting from the character at index1 and till end of string |
| **S[:index2]** | Returns a substring of s starting from the character at beginning of string and till character at index2-1 |
| **w in s** | Return a Boolean value indicating whether w is a substring in s |
| **Str(number)** | Converts a number into a string |

| NOTE |
|---|
| Nest ‘’ and “” when you want to explicitly print the quotation mark e.g.<br>*In >> Print (' "car" ')*<br>*Out >> “car”* |

# Lists

| | |
|---|---|
| **L[i]** | Access an element in list L whose index is i, i.e. access the i’th element of list |
| **L[-i]** | Access the element in list L whose index is i from the end of the list, i.e. L[-1] returns the last element of the list |
| **L[i] = X** | replace the element in the list at index i with x |
| **L.append(x)** | Add element x to the end of the list L |
| **L.insert(i, x)** | Add element x to the list at the index i |
| **del L[i]** | Delete the i’th element in the list |
| **L.pop(i)** | Delete and return the i’th element from list L |
| **L.pop()** | Delete and return the last element element from list L |
| **L.remove(x)** | Remove element x from the list L |
| **L[i1:i2]** | Returns a sublist of L starting from the element at i1 and ending at i2-1 |
| **L[i1:]** | Returns a sublist of L starting from the element at i1 and till end of string |
| **L[:i2]** | Returns a sublist of L starting from the element at beginning of List and till the element at i2-1 |
| **L[:]** | Returns a copy of list L |
| **x in L** | Returns true if element x is in the list |
| **X not in L** | Returns true if element x is not in the list |
| **L.sort()** | Sorts list L in ascending order, modifies original list |
| **L.sort(reverse = True)** | Sorts list L in descending order, modifies original list |
| **sorted(L, reverse = True)** | Returns a copy of list L sorted in descending order |
| **sorted(L)** | Returns a copy of list L sorted in ascending order |

| | |
|---|---|
| **L.reverse()** | Modifies list L and reverses the order of its element so that the last element is the first one etc. |
| **len(L)** | Returns the number of elements in the list L |
| **min(L)** | Returns the minimum of list L |
| **max(L)** | Returns the maximum of list L |
| **Sum(L)** | Returns the sum of all elements in a list of numbers |

| NOTE |
|---|
| List indexing starts from 0 not from 1, i.e. the first element of the list is accessed by list[0] |

## Comprehension List

A **list comprehensio**n allows you to generate list in just one line of code. e.g.

- List = [value**2 for value in **range**(1,11)] is a list of squares of numbers between 1 and 11

## Tuples

- A tuple contains an ordered collection of values or items
- Tuples are defined as a comma-separated list of item
- They can contain elements of the same or different data types
- Tuples can be accessed in same manner as lists
- Tuples are immutable
- Most functions that are applicable to lists are also applicable to tuples as long as they don't attempt to modify the tuple
- It doesn't support methods like append(), insert(), delete()
- We cannot assign values to elements of tuples by indexing them

## Dictionaries

| | |
|---|---|
| **dict[key]** | Returns the value in the dictionary dict associated with key |
| **dict[new_key] = value** | Adds a new key and value pair to the dictionary dict |
| **dict[key]=value** | Modify the value associated with an existing key in the dict |
| **del dict[key]** | Delete key value pair from dictionary dict |
| **dict.keys()** | Returns the entire list of keys in the dictionary dict in no particular order |
| **sorted(dict.keys())** | Returns a sorted list of keys in the dictionary dict |
| **dict.items()** | Return a list of tuples representing the keys & values; used to loop over dictionary |
| **dict.values()** | Returns the entire list of values in the dictionary dict in no particular order |
| **set(dic.values()):** | Returns a *unique* list of values in the dictionary dict |
| **[dict1, dict2]** | Create a list of dictionaries |
| **{ key:[v1, v2, v3] key2: [v]}** | Create a dictions whose values are a list of elements |
| **{key: {key:value, key:value}}** | Nest a dictionary inside a dictionary |

| NOTE |
|---|
| It's good practice to include a comma after the last key-value pair as well, so you're ready to add a new key-value pair on the next line. |

## If Condition

| SYNTAX |
|---|
| **if** <condition>:<br>  <expr><br>**elif** <condition>:<br>  <expr><br>**else:**<br>  <expr> |

| | |
|---|---|
| **<condition> and <condition>** | Boolean condition validating that both condition are true |
| **<condition> or <condition>** | Boolean condition validating that either or both conditions are true |
| **==, !=, <=, >=, <, >** | Boolean operators used to check for equality, inequality, less than or equality, greater than or equality condition |
| **If <list_name>:** | Return true if the list list-name has at least one element |
| **True** | Boolean literal case sensitive |
| **False** | Boolean literal that is case sensitive |

| NOTE |
|---|
| Only one block is executed in an if -elif -else chain<br>you can use parentheses around the individual conditional tests, but they are not required |

## For Loops

| SYNTAX |
|---|
| **For** <expr> **in** <expr>:<br>  <expr> |

- Every *indented* line following the line for is considered inside the loop
- Expressions within a loop are executed repeatedly until the expression <expr> in <expr> evaluates to False
- Non indented lines are executed once without repetition as they are considered outside the for loop
- You shouldn't modify a list using a for loop because Python will have trouble keeping track of the items in the list (use while loop)

## While Loop

| SYNTAX |
|---|
| **While** <condition>:<br>  <expr> |

| | |
|---|---|
| **break** | to exit a while loop immediately |
| **continue** | to ignore the rest of the loop and return to the beginning of loop |

- Indented lines following the while loop are executed repeatedly as long as the while condition is true
- To modify a list as you work through it, use a while loop
  - e.g. while <list-name> : loops until list is empty even if it is modified within loop

# Functions

| SYNTAX |
| --- |
| **def <fun_name>(**<parameter>**):**<br>  <expr> |

| | |
| --- | --- |
| """ **function description** """ | Docstring describes what the function does used generate |
| **return** <value> | Return a value to the calling function |
| **def** fun-name**(**parameter = default_value**):** | Set a default value for a function parameter |
| **def** fun-name**(**parameter = ''**):** | Set an optional parameter |
| **def** fun-name**(***parameter**):** | Set an arbitrary parameter a tuple is created containing the arguments passed |
| **def** fun-name**(****parameter**):** | Set an arbitrary key value pair, creates a dictionary containing the key value arguments passed |

- TWO ways to pass arguments to functions
  - ***Positional argument***, where position of parameter matters
    - fun-name**(**parameter1, param2, etc**)** assigned by order
    - mixing up the order gives unexpected results
  - ***Key word argument***, where each argument consists of a variable name and value i.e. key value pair
    - fun-name(parameter-name = args, parameter-name = args )
    - order of argument doesn't matter as each parameter is explicitly assigned to its argument
- Functions modify permanently a parameter list therefore, use a copy of the list to avoid changing the original list
- Default, optional and arbitrary parameters should be placed at the end of list of parameters
- Positional arguments, keyword arguments, default, and optional values can all be used together

# Classes

| SYNTAX |
| --- |
| **class** <Name>**():**<br>""" **class description** """<br>  **def __init__(self**, parameter**)**<br>  **self.**<attribute> = <value><br>  **def get**_attribute(self):<br>  **return self**.attribute<br>  **def update**_attribute(self):<br>  **self.**attribute **=** new_value |

| | |
| --- | --- |
| """ **class description** """ | Docstring describes what the class does |
| **def __init__(self**, parameter**)** | This method is run automatically and return an instance of class |

| **<var_name> = className()** | Create instance of class name |
|---|---|

- Self is a required parameter that must be the first parameter in an instance method, it allows accessing and modifying the class instance
- Every method call associated with a class automatically passes the **self** object i.e. a reference to the current object
- Every attribute in a class needs an initial value, default values must be specified in __init__ function
- To modify an instance attribute:
  - directly Instance_name.attribute_name = new value
  - Update the value through an update method instance_name.update_atribute(new)

# Inheritance

| SYNTAX |
|---|
| **Class <parent_name>():**<br><br>**class** <Name>**(**parent_inheritance_class_name**):**<br>**def __init__(self**, parameter**)**<br>**self**.< attribute > = <value> |

- The parent class must be part of the current file and must appear before the child class in the file.
- __init__() method takes in the same attribute as parent init method and calls super().__init__()??

# Modules

| | |
|---|---|
| **import** module_name | Import module<br>• Access classes using dot notation module e.g. module_name.class_name.function_name |
| **Import** module_name * | Import every class in module<br>• it's unclear which classes you're using from the module<br>• potential naming conflict<br>• can access without dot notation |
| **import** module_name **as** mn | Import a module with an alias mn<br>• Respective calls to methods/class in this module must be accesed using the alias name mn.class<br>• avoid name conflict or if name is too long |
| **from** module_name **import** class_name, class_name | Import multiple classes from a module<br>• Can access class directly |
| **from** module_name **import** function_name, function_name, etc | Import a specific function from a module<br>• Can use function directly |
| **from** module_name **import** function_name **as** fn | Import a function from module with an alias fn<br>• Can use function directly using its alias name<br>• Avoid name conflict or if name is too long |
| **from** module_name **import** * | Import every class in module<br>• can call each function by name without using the dot notation<br>• Not advisable for large modules; mixing names |

| | |
|---|---|
| | • If same function names exist in current class they will be overridden |

## Basics

| | |
|---|---|
| **#** comment | Single line comment in python starts with a hash |
| **None** | Equivalent to null in Java |
| **choice**([value1, value2]) | Returns randomly either value1 or value2 |
| **range(**value1, value2**)** | Returns a list of numbers starting from value 1 to value2 -1 with increment of 1 between each element of list |
| **range(**value1, value2, increment**)** | Returns a list of numbers starting from value1 to value2-1 with increment specified by the parameter |
| Number1 **number2 | Returns number1 to the exponent of number 2 |

| NOTE |
|---|
| Python uses indentation to determine when one line of code is connected to the line above, improperly indented code generates errors<br>Link to standard library https://pymotw.com/3/ |

## Convenions PEP 8 guideline

- Convention documentation https://www.python.org/dev/peps/pep-0008/
- Each line should be less than 80 characters
- Limit comments to 72 characters per line
- Indentation is equivalent 2 four spaces, don't use tabs use always spaces
- Use blank lines to organize your files, use two empty lines to separate functions
- Use descriptive names with lower case and underscore
- Functions should always have docstring explaining what they do
- Classes should have a docstring immediately following the class definition
- Modules should have a docstring in the first line describing its
- For default parameters, no spaces should be used on either side of the equal sign
- If function parameters span extra lines use two tabs instead of one to distinguish them from the body of the function
- All import statements should be written at the beginning of a file
- Class names should capitalize the first letter of each word in the name, and don't use underscores e.g. CarEngine
- Use one blank line to separate methods in a classes
- Use two blank lines to separate classes in a module
- Use one blank line to separate Import module statements that are created by you than import statements for other modules
- Convention on naming getters and setters methods in a class
    - **get**_name_atr
    - **update**_name_attr

## User Input

- **Input(**string_instruction**)** Prints the corresponding instruction to command line and pauses the program waiting for user input which is returned when user hits enter
- Python interprets everything the user enters as a string.
- Use int() to change from str to int

## Read File

| SYNTAX | |
|---|---|
| **With open(**<file_path>, <mode>**) as** file_object**:**<br><expr> | |
| **For** line **in** file_object **:**<br><expr> | |
| **With open(**<file_path>, **'r')** | Returns an object representation of the file in read mode |
| **With open(**<file_path>, **'w')** | Returns an object representation of the file in write mode, erases existing content of file |
| **With open(**<file_path>, **'a')** | Returns an object representation of the file in append mode, text is added to end of the file |
| **With open(**<file_path>, **'r+')** | Returns an object representation of the file in read & write mode |
| File_object**.read()** | Returns the entire content of the file as a string |
| file object**.readLines**() | Returns a list of lines in the file object |

- You can use either absolute or relative paths for accessing files
- **With** clause closes the file once access to it is no longer needed
- Use rstrip() to remove to extra blank line
- By default file is opened in read-only mode if mode is omitted
- When writing to file always add new line at end

## Exception

| SYNTAX |
|---|
| **try:**<br><expr><br>**except** <exception_object_type**>:**<br><expr> / **pass** |
| **try:**<br><expr><br>**except** <exception_object_type**>:**<br><expr> / **pass**<br>**else:**<br><expr> |

- **Pass** is a form of failing silently as no actions are done if exception is found
- The **else** block must be executed if the try clause does not raise an exception
- The else clause avoids accidentally catching an exception that wasn't raised by the code being protected by the try. The except block must only contain the corresponding statement

## JSON

| | |
|---|---|
| **Import json** | Import the json module |
| **json.dump(**object, file**)** | Dump simple python data structures into a file |
| **Json.load(**file**)** | Load the data stored in the corresponding file |

- JSON is a comma separated string
- Allows you to dump simple Python data structures into a file and load the data from that file
- json.dump(object, file) function takes two arguments: a piece of data to store and a file object it can use to store the data

# Unitest

SYNTAX

```
Import unittest
# import the functionality to test

Class funtestcase(unittest.TestCase): # inherit fro, unitest.Testcase
        def setUp(self):
                # create objects to test these will be shared across all unittest functions
                # runs before running each method whose name starts with test
                self.var_name = new_instance
        def test_function_name(self):
                Self.assertequal(result, expected value)

unittest.main() # tell python to run the file
```

| | |
|---|---|
| **assertEqual**(a, b) | Verify that a == b |
| **assertNotEqual**(a, b) | Verify that a != b |
| **assertTrue**(x) | Verify that x is True |
| **assertFalse**(x) | Verify that x is False |
| **assertIn**(item, list) | Verify that item is in list |
| **assertNotIn**(item, list) | Verify that item is not in list |

- Automate testing
  - A ***unit test*** verifies that one specific aspect of a function's behavior is correct
  - A ***test case*** is a collection of unit tests that together prove that a function behaves correct
- A test case with full coverage includes a full range of unit tests covering all the possible inputs for a method
- Any method that starts with test_ will run automatically
- Python prints a single character for each unit test
  - A dot for a passing test
  - An E for a test resulting in error, and
  - An F for a failing assertion