Python - Essentials

Python

- Python is a high-level, interpreted, interactive and object-oriented scripting language.

Variables

- Variables are containers for storing data values.
- Unlike other programming languages, Python has no command for declaring a variable.

```
x = 5
y = "John"
print(x)
```

Data Types

- Some of data types in Python
  - Numbers `int, float, complex
  - String `str`
  - List `list`
  - Tuple `tuple`
  - Set `set`
  - Dictionary `dict`
  - Boolean `bool`
  - None `NoneType`

Numbers

Numbers in python are are represented by `int`, `float`, `complex`

- `int` : integer `1, 2, 3, 4, 5`
- `float` : floating point number `1.0, 2.0, 3.0, 4.0, 5.0`
- `complex` : complex number `1+2j, 2+3j, 3+4j, 4+5j, 5+6j`
- `type(5)` -> `int`
- `type(5.0)` -> `float`
- `type(5+6j)` -> `complex`

Strings

Strings are arrays of bytes representing unicode characters, `str`

- `str` : string `"hello", "world", "python", "programming"`
- `type("hello")` -> `str`
- `type('hello')` -> `str`
- `type("""hello""")` -> `str`

Lists

Lists are used to store multiple items in a single variable, `list`

- `list` : list `[1, 2, 3, 4, 5]`
- `type([1, 2, 3, 4, 5])` -> `list`
- `type([1, 2.0, 3+4j, "hello", [1, 2, 3]])` -> `list`
- `type([])` -> `list`

some common list methods

list -> `[2,'qasim', 3, 4, 5]`

- `len([2,'qasim', 3, 4, 5])` -> `5`
- `list.append(6)` -> `[2,'qasim', 3, 4, 5, 6]`
- `list.remove('qasim')` -> `[2, 3, 4, 5, 6]`
- `list.pop()` -> `[2, 3, 4, 5]`
- `list.insert(1, 'qasim')` -> `[2, 'qasim', 3, 4, 5]`
- `list.sort()` -> `[2, 3, 4, 5, 'qasim']`
- `list.reverse()` -> `['qasim', 5, 4, 3, 2]`
- `list.clear()` -> `[]`
- `list.copy()` -> `[2, 'qasim', 3, 4, 5]`
- `list.count(2)` -> `1`

Tuples

Tuples are used to store multiple items in a single variable, they are different from lists in the sense that they are immutable i.e they cannot be changed, `tuple`

- `tuple` : tuple `(1, 2, 3, 4, 5)`
- `type((1, 2, 3, 4, 5))` -> `tuple`
- `type((1, 2.0, 3+4j, "hello", [1, 2, 3]))` -> `tuple`

Some common tuple methods

tuple -> `(2,'qasim', 3, 4, 5)`

- `len((2,'qasim', 3, 4, 5))` -> `5`
- `tuple.count(2)` -> `1`
- `tuple.index(2)` -> `0`
- `tuple[0]` -> `2`
- `tuple[1]` -> `qasim`>
- `tuple[2]` -> `3`

Sets

Sets are used to store multiple items in a single variable, they are unordered and unindexed, `set`, repeated items are not allowed

- `set` : set `{1, 2, 3, 4, 5}`

- `type({1, 2, 3, 4, 5})` -> `set`
- `type({1, 2.0, 3+4j, "hello", [1, 2, 3]})` -> `set`

Some common set methods

set -> `{2,'qasim', 3, 4, 5}`

- `len({2,'qasim', 3, 4, 5})` -> `5`
- `set.add(6)` -> `{2,'qasim', 3, 4, 5, 6}`
- `set.remove('qasim')` -> `{2, 3, 4, 5, 6}`
- `set.pop()` -> `{2, 3, 4, 5}`
- `set.clear()` -> `set()`
- `set.copy()` -> `{2, 'qasim', 3, 4, 5}`
- `set.discard(2)` -> `{'qasim', 3, 4, 5}`

Dictionaries

Dictionaries are used to store data values in key:value pairs, `dict`

- `dict` : dict `{"name": "qasim", "age": 25, "city": "Ottawa"}`
- `type({"name": "qasim", "age": 25, "city": "Ottawa"})` -> `dict`
- `type({"name": "qasim", "age": 25, "city": "Ottawa", "hobbies": ["coding", "reading", "gaming"]})` -> `dict`

Some common dictionary methods

dict -> `{"name": "qasim", "age": 25, "city": "Ottawa"}`

- `len({"name": "qasim", "age": 25, "city": "Ottawa"})` -> `3`
- `dict["name"]` -> `qasim`
- `dict.get("name")` -> `qasim`
- `dict.keys()` -> `dict_keys(['name', 'age', 'city'])`
- `dict.values()` -> `dict_values(['qasim', 25, 'Ottawa'])`
- `dict.items()` -> `dict_items([('name', 'qasim'), ('age', 25), ('city', 'Ottawa')])`
- `dict.pop("name")` -> `qasim`

Booleans

Booleans represent one of two values: `True` or `False`

- `bool` : bool `True, False`
- `type(True)` -> `bool`
- `type(False)` -> `bool`

Some common boolean methods

- `bool(0)` -> `False`
- `bool(1)` -> `True`
- `bool(2)` -> `True`

- `bool(3)` -> `True`

Comparison Operators

- `==` : equal
- `!=` : not equal
- `>` : greater than
- `<` : less than
- `>=` : greater than or equal to
- `<=` : less than or equal to

Logical Operators

- `and` : and
- `or` : or
- `not` : not
- `&` : and
- `|` : or
- `~` : not

if, elif, else Statements

```
if condition:
    # code
elif condition:
    # code
else:
    # code
```

for Loops

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

for val in "string":
    if val == "i":
        break
    print(val)
```

while Loops

```
i = 1
while i < 6:
    print(i)
```

```
    i += 1
while True:
    print("hello")
    break
```

list comprehension

list comprehension is a way to create lists in python

```
new_list = [i for i in old_list]
new_list = [i.upper() for i in old_list]
new_list = [i for i in old_list if i != 0]
```

functions

functions are a block of code that only runs when it is called there are many ways to define a function

```
def my_function(name):
    print(f"hello {name}")
def my_function(name="qasim"):
    print(f"hello {name}")
def my_function(*args):
    while args:
        print(args.pop())
def my_function(**kwargs):
    for key, value in kwargs.items():
        print(f"{key} : {value}")
```

lambda expressions

lambda expressions are used to create anonymous functions and are used to create small functions

```
x = lambda a : a + 10
print(x(5))

my_function = lambda a : a + 10
print(my_function(5))
print((lambda a : a + 10)(5))
```

map and filter

map and filter are used to apply a function to a list of elements map will return a list of the results after applying the given function to each item of a given iterable filter will return a list of the elements of the iterable for which the function returns True

```
my_list = [1, 2, 3, 4, 5]
new_list = list(map(lambda x: x * 2, my_list))
new_list = list(filter(lambda x: x % 2 == 0, my_list))
```

methods

- `dir()` : returns a list of all the methods and properties of the object
- `help()` : returns the documentation of the object
- `type()` : returns the type of the object
- `id()` : returns the unique id of the object

nested statements and scope

- LEGB Rule
    - L : Local
    - E : Enclosing
    - G : Global
    - B : Built-in the scope of a variable is the context in which it is defined.

```
x = 300
def my_function():
    x = 200
    print(x)
print(x) # 300
def my_function():
    global x
    x = 200
    print(x) # 200
```

file I/O

to use files in python, you have to open them first, using the built-in open() function

```
f = open("demofile.txt", "r")
print(f.read())
f.close()
```

REPL

- Read : read the user input
- Evaluate : evaluate the user input
- Print : print the result
- Loop : loop the above process until the user quits

helpful funvtions

- type()
- dir()
- help()

String functions

- len()
- lower()
- strip()
- split()

List functions

- append()
- remove()
- set()
- sort()
- join() `" * ".join(list)` -> `a * b * c` where list = [a, b, c]

list single line for loop

- `new_list = [i.upper() for i in old_list]`
- `set_list = {i.upper() for i in old_list}`
- `dict_list = {i: i.upper() for i in old_list}`

creating copy of list

- `new_list = old_list` # here the changes in new_list will reflect in old_list
- `new_list = old_list[:]` # here the changes in new_list will not reflect in old_list
- `new_list = old_list.copy()` # here the changes in new_list will not reflect in old_list
- `new_list = list(old_list)` # here the changes in new_list will not reflect in old_list

zip function

- `zip([1, 2, 3], [4, 5, 6])` -> `[(1, 4), (2, 5), (3, 6)]`
- zip combine the elements of two lists in a tuple

Packages

Standard Library

- `import random`
- `import math`
- `import os`
- `import sys`
- `import json` full list of standard library: https://docs.python.org/3/library/

Creating and Using module

- cerate folder `mkdir my_module`
- create file `touch my_module/my_module.py`
- create file `touch my_module/__init__.py`
- `from my_module import my_module`
- `my_module.my_function()`

os

- `os.getcwd()` gets current working directory
- `os.chdir('path')` change directory
- `os.listdir()` list all files and folders in current directory

sys

- `sys.path` list of all the directories where python looks for modules
- sys.argv is a list in Python, which contains the command-line arguments passed to the script.
- `sys.exit()` exit from python

basic command line arguments

- `python3 my_script.py arg1 arg2 arg3`

```
import sys

print('Number of arguments:', len(sys.argv), 'arguments.')
```

Testing in Python

- unit testing : testing individual units or components of a software
- integration testing : testing the combination of two or more units
- functional testing : testing the functionality of the software
- end-to-end testing : testing the complete flow of the software

assert

- `assert 1 == 1` # no output
- `assert 1 == 2` # AssertionError

unittest

- `import unittest`
- `unittest.TestCase` is a class that allows us to create test cases

Sample unittest

```

```

```
import unittest
class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())
```

asserts

- assertEqual(a, b) checks a and b are equal
- assertNotEqual(a, b) checks a and b are not equal
- assertTrue(x) checks x is True
- assertFalse(x) checks x is False
- assertIs(a, b) checks a is b
- assertIsNot(a, b) checks a is not b
- assertIsNone(x) checks x is None
- assertIsNotNone(x) checks x is not None

Web Frameworks

- Django: high-level web framework
- Flask: micro web framework
- Pyramid: open source web framework
- Bottle: simple web framework

Flask

- `pip install flask`
- making a simple hello world with welcome animated message

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

- `flask run` to run the server, it will look for `app.py` and run the server on port : 5000
- if the file name is different then `FLASK_APP=script.py flask run`

Routess with parameters

```
@app.route('/user/<username>')
def show_user_profile(username):
```

```
    # show the user profile for that user
    return 'User %s' % username
```

Templates

- we can pass arguments to the templates
- `render_template('index.html', name=name)`
- the template file should be in `templates` folder
- inorder to render the passed argument in the template file we use `{{ name }}`

```
@app.route('/hello/<user>')
def hello_name(user):
    return render_template('hello.html', name=user)
```

```
<h1> welcome {{ name }} </h1>
```

Templating syntax

- `{{ variable }}` : to print the variable
- `{% for i in list %} {{ i }} {% endfor %}` : for loop
- `{% if condition %} {{ variable }} {% endif %}` : if condition
- `{% block content %} {% endblock %}` : block of content
- `{% extends "layout.html" %}` : extending the layout file
- `{% include "header.html" %}` : including the header file

Template syntax - 2

- `{% set name = "qasim" %}` : setting the variable
- `{% macro render_title(title) %} <title>{{ title }}</title> {% endmacro %}` : creating a macro
- `{{ render_title("Home") }}` : using the macro
- the diffrence between `{{ }}` and `{% %}` is that `{{ }}` is used to print the variable and `{% %}` is used to write the logic

the `requests` module

- requests module is used to make the http requests
- `pip install requests`

```
import requests

@app.route('/repos/<username>')
def repos(username):
```

```
    r = requests.get(f'https://api.github.com/users/{username}/repos')
    return r.json()
```

the `requests` module - 2

```
@app.route('/repos/<username>')
def repos(username):
    r = requests.get(f'https://api.github.com/users/{username}/repos')
    return render_template('repos.html', repos=r.json())
```

```
<ul>
    {% for repo in repos %}
        <li>{{ repo.name }}</li>
    {% endfor %}
</ul>
```

calling other routes from the template

```
<form action="/repos" method="get">
    <input type="text" name="username">
    <input type="submit" value="Submit">
</form>
```