Python Basics Review

Use libraries

Introduce `module`, `function`, `constants` and others into the current Python script.

```
import [MODULE NAME] as [NICKNAME]
from [MODULE NAME] import [SUBMODULE NAME | FUNCTION NAME | OTHER NAME]
from [MODULE NAME] import *
```

Functions

How to formally define a function

- **Step 1**: Provide and fix the function signature (i.e. "function name" + "function arguments including types of the arguments" + "function returns and the types of returns").
- **Step 2**: Clarify the domain of each function argument. By "domain", it means the set of values that are valid for this argument.
  For example, given a function "f(x, y) = x/y", the domain of y is all real numbers except 0.
- **Step 3**: Clarify the range of each function return. By "range", it means the set of all possible and valid values for this return.
  For example, given a function "f(x, y) = x/y", the return of f(x, y) is the set of all real numbers.
- **Step 4**: Write down the function signature (as a part of code), the function arguments and their domains as well as the function returns and their ranges (into the function description as a part of document).
- **Step 5**: Create a set of test cases to test if your function implementation is correct.
  For example, for the function "f(x, y) = x/y", I create:
  - Test Case #1: x = 1.2, y = 0.2, f(x, y) = 6
  - Test Case #2: x = 0.1, y = 0, throw exception of "divided by zero"
  - Test Case #3: x = 0, y = 100, f(x, y) = 0
- **Step 6**: Implement the function.
- **Step 7**: Test the function using the defined test cases.

Function definition format

```
def [FUNCTION NAME]([ARGUMENT LIST]):
    [FUNCTION BODY]
```

Notations for types are supported but not interpreted. Type notations are mostly defined in the built-in module `typing`.

```
def func(x: int) -> Integer
```

CAUTION

Types in Python code usually are implicit. Be careful about types of inputs and outputs.

Main function

```
if __name__ == '__main__':
    ...
```

Exceptions

- Try/Catch

```
try:
    [TRY BODY]
except Exception as e:
    [EXCEPTION CASE BODY]
else:
    [NOTHING WRONG BODY]
finally:
    [RUN ANYWAY BODY]
```

- Throw exceptions

```
raise Exception([EXCEPTION CONTENT])
```

Primitive Types

- String: `str`

```
x = 'hello!'
```

- Integer: `int`

```
x = 1
```

- Float: `float`

```
x = .16
```

- Complex: `complex`

```
x = 0.5 + 2j
```

- Bool: `bool`

```
x = True
```

- List: `list`

```
x = ['a', 'b', 'c']
```

- Tuple: `tuple`

```
x = ('ID', 2, 0.5)
```

- Range: `range`

```
x = range(1, 10)
```

- Set: `set`

```
x = {1, 2, 3}
```

- FrozenSet: `frozenset`
  The set is immutable.

```
x = frozenset({1, 2, 3})
```

- NoneType: `None`

```
x = None
```

- Bytes: `bytes`

```
x = bytes('abc')
```

- MemoryView: `memoryview`

  Expose the memory of a `bytes` variable.

  ```
  x = memoryview(bytes('abc'))
  ```

Variables

```
[VARIABLE NAME] = [VARIABLE VALUE | EXPRESSION]
```

Arithmetic Operations

- Addition

  ```
  x = x + 1
  x += 1
  ```

- Subtraction

  ```
  x = x - 1
  x -= 1
  ```

- Multiplication

  ```
  x = x * 1
  x *= 1
  ```

- Division

  ```
  x = x / 1
  x /= 1
  ```

- Modulus

  ```
  x = x % 2
  x %= 2
  ```

- Exponentiation

  ```
  x = x ** 2
  x **= 2
  ```

- Floor Division

```
x = x // 2
x //= 2
```

Logic Operations

- AND

  ```
  x and y
  ```

- OR

  ```
  x or y
  ```

- NOT

  ```
  not x
  ```

Condition Operators

- Equal

  ```
  x == y
  ```

- Not Equal

  ```
  x != y
  ```

- Less Than

  ```
  x < y
  ```

- Less Than Or Equal To

  ```
  x <= y
  ```

- Greater Than

  ```
  x > y
  ```

- Greater Than Or Equal To

  ```
  x >= y
  ```

- Ternary Operator

  ```
  [VALUE ON TRUE] if [CONDITION EXPRESSION] else [VALUE ON FALSE]
  ```

  For example,

  ```
  x = 1
  y = 2
  min_val = x if x < y else y
  ```

  `min_val` is `1` from this code.

Conditional Statements

```
if [CONDITION EXPRESSION]:
    [STATEMENTS]
elif [CONDITION EXPRESSION]:
    [STATEMENTS]
else:
    [STATEMENTS]
```

Loops

- While loops

  ```
  while [LOOP CONDITION]:
      [LOOP BODY]
  ```

- For loops

  ```
  for [EACH ELEMENT] in [ITERABLE OBJECT]:
      [LOOP BODY]
  ```

- Break: `break`
- Continue: `continue`

List Operations

- Conversion: `list()`

  ```
  x = list(range(1, 10))
  ```

- Empty List: `[]`

```
x = []
```

- Length: `len()`

```
x = len([1, 2, 3])
```

- Append: `append()`

```
x = [1, 2, 3]
x.append(4)
```

- Concatenate: `+`

```
x = [1, 2, 3] + [4, 5]
```

- Sub-list (i.e. slicing):

```
x = [1, 2, 3, 4, 5]
x[1]
x[1:3]
x[:3]
x[-1]
x[2:]
x[:]
```

- Modification:

```
x = [1, 2, 3, 4, 5]
x[1] = 10
x[1:3] = [11, 12]
x[1:3] = [21, 22, 23, 24]
```

- Composition

```
l = [x for x in range(1, 11)]
l = [x for x in range(1, 11) if x % 2 == 0]
l = ['E' if x % 2 == 0 else 'T' if x % 3 == 0 else x for x in range(1, 11)]
```

- There are more operations such as `extend()`, `insert()`, `remove()`, `pop()`, `reverse()`, `min()`, `max()`, `count()`, `sort()`, `index()` and `clear()`.

Set Operations

- Conversion: `set()`

```
x = set([1, 2, 3])
x = set((1, 2, 3))
```

- Empty Set: `set({})`

```
x = set({})
```

- Length: `len()`

```
x = len({1, 2, 3})
```

- Union: `union()` or `|`

```
x = {1, 2, 3}.union({4, 5})
```

- Intersection: `intersection()` or `&`

```
x = {1, 2, 3}.intersection({1, 4, 5})
```

- Difference: `difference()` or `-`

```
x = {1, 2, 3}.difference({1, 4, 5})
```

- Symmetric Difference: `symmetric_difference()` or `^`

```
x = {1, 2, 3}.symmetric_difference({1, 4, 5})
```

- Is Disjoint: `isdisjoint()`

```
x = {1, 2, 3}.isdisjoint({1, 4, 5})
```

- Is Subset: `issubset()` or `<=`

```
x = {1, 2, 3}.issubset({1, 4, 5})
```

- Is Proper Subset: `<`

```
x = {1, 2, 3} < {1, 4, 5}
```

- Is Superset: `issuperset()` or '>='

```
x = {1, 2, 3}.issuperset({1, 4, 5})
```

- There are more operations such as `remove()`, `update()`, `discard()`, `pop()` and `clear()`.

Dict Operations

- Length: `len()`

```
x = len({'a':1, 'b':2})
```

- Empty Dict: `{}`

```
x = {}
```

- Get Items: `items()`

```
x = {'a':1, 'b':2}
x.items()
```

- Enumerate and Unpack Items: `enumerate()`

```
x = {'a':1, 'b':2}
enumerate(x.items())
```

- Get Keys: `keys()`

```
x = {'a':1, 'b':2}
x.keys()
```

- Get Values: `values()`

```
x = {'a':1, 'b':2}
x.values()
```

- Determine if a key is in Dict: `in`

```
x = {'a':1, 'b':2}
'c' in x
```

- Access value for a given key:

```
x = {'a':1, 'b':2}
x['b']
```

- Remove an item: `del`

```
x = {'a':1, 'b':2}
del x['a']
```

- Remove an item and return the value: `pop()`

```
x = {'a':1, 'b':2}
val = x.pop('a')
```

- There are other operations such as `clear()`, `copy()`, `fromkeys()`, `get()`, `pop()`, `popitem()`, `setdefault()` and `update()`.

String Operations

- Length: `len()`

```
x = len('abc')
```

- Concatenation: `+`

```
x = 'hello ' + 'world!'
```

- As List:

```
x = 'abcd'
x[1]
x[-1]
x[1:3]
```

- Empty String: `''`

```
x = ''
```

Classes

Define a class

```
class [CLASS NAME]:
    [CLASS BODY]
```

Inheritance

```
class [CLASS NAME]([PARENT_1 NAME], [PARENT_2 NAME], ...):
    [CLASS BODY]
```

"Self" for instance: `self`

- `self` will be set as the first argument to each non-static member function of a class.
- To access member functions and member variable, we need to use `self`.

```
self.member_func()
```

"Constructor"

```
class [CLASS NAME]([PARENT_1 NAME], [PARENT_2 NAME], ...):
    def __init__(self, [ARGUMENTS]):
        [CONSTRUCTOR BODY]
```

Polymorphism

```
class Parent:
    def poly_func():
        [PARENT IMPLEMENTATION]

class Child(Parent):
    def poly_func():
        [Child IMPLEMENTATION]
```

Invoke Parent's functions: `super()`

```
class Parent:
    def poly_func():
        [PARENT IMPLEMENTATION]

class Child(Parent):
    def poly_func():
        super().poly_func()
        [Child IMPLEMENTATION]
```

Public, Protected, and Private Member Variables and Methods

- In general, a **public** variable/function is accessible to any instance of this class as well as any child class.
- In general, a **protected** variable/function is only accessible to any child class.
- In general, a **private** variable/function is only accessible inside the class definition.
- The above three general semantics may vary in different programming languages.
- `_` in front of a member variable/function's name makes it **protected**.

- `__` in front of a member variable/function's name makes it **private**.

```
class MyClass:
    def public_func(self):
        ...

    def _protected_func(self):
        ...

    def __private_func(self):
        ...
```

Static member functions: `@classmethod , cls`

- `@classmethod` annotation needs to be used.
- `cls` will be the first argument for static function.
- Invoking static function is done without creating class instances.

```
class MyClass:
    @classmethod
    def static_func(cls):
        [IMPLEMENTATION]

MyClass.static_func()
```

Method Resolution Order (MRO)

- The invoking of a function from a class that has no implementation will trigger a search of this function in the ancestors of the class in a certain order of the ancestors. This order is called **MRO**. And the search stops when the first implementation of the function is found in the MRO.
- Use `__mro__` to access MRO.

```
class Child(Parent_1, Parent_2):
    ...
Child.__mro__
```

Access immediate parents: `__bases__`

```
class Child(Parent_1, Parent_2):
    ...
Child.__bases__
```

Create class instances

```
class MyClass:
    ...
myclass_ins = MyClass()
```

Access ID of an object: `id()`

```
class MyClass:
    ...
myclass_ins = MyClass()
id(myclass_ins)
```

Multitasking

- Multiprocessing: `multiprocessing`
- Multithreading: `threading`

Async/Await: `asyncio`

- Define async function:

```
async def [FUNCTION NAME]([ARGUMENTS]):
    [FUNCTION BODY]
```

- Invoke await functions:

```
async def [FUNCTION NAME]([ARGUMENTS]):
    ...
    await asyncio.[FUNCTION]
```

- Create await tasks:

```
async def [FUNCTION NAME]([ARGUMENTS]):
    ...
    task = asyncio.create_task([ARGUMENTS])
    await task
```