Contents

1 Basics

1.1 Output to Console

```
print('Hello World')
```

```
Hello World
```

By default print function will output the next `print` function in the next line.

```
print('Hello')
print(' World!')
```

```
Hello
 World!
```

Adding `end` parameter changes the last character in the print function. The default `end` is `\n`.

```
print('Hello', end='')
print(' World!')
```

```
Hello World!
```

1.2 Getting Input from the Console

Getting input from the console and passing them to a variable

```
print('Give me your name')
name = input()

print ('Your name is ' + name )
```

```
Give me your name
```

Get input without a print statement

```
name = input('Give me your name: ')

print('Your name is ' + name)
```

```
Give me your name:
```

Convert string to float or int

```
age = input('What is your age') # input() returns a string
age = int(age) # 'age' is now an integer
age = age + 5
print('Your age will be ' + age + ' in the next five years' )
```

```
What is your age
```

1.3 String Formatting

String formatting is the process of infusing things in the string dynamically and presenting the string.

1.3.1 % Operator Formatting.

Using the % format tells Python to substitute the value of the `name` variable.

```
name = 'Slim Shady'
print ('My name is %s' % name)
```

```
My name is Slim Shady
```

1.3.2 str.format()

You can use `format()` to do simple positional formatting

```
name = 'Slim Shady'
print ('My name is {}'.format(name))
```

```
My name is Slim Shady
```

Or you can refer to your variable substitutions by name and use them in any order.

You can use `format()` to do simple positional formatting

```
name = 'Yoshikage Kira'
age  = 33
print ('My name is {name}, I am {age} years old.'.format(age=age,name=name))
```

```
My name is Yoshikage Kira, I am 33 years old.
```

1.3.3 f-strings

NOTE: This only works in Python **version 3.6 or later.**

This new way of formatting lets you use embedded python expressions inside string constants.

```
name = 'Yoshikage Kira'
age  = 30
print(f'My name is {name}. I am {30 + 3} years old')
```

```
My name is Yoshikage Kira. I am 33 years old
```

2 Variables

Variables are containers for storing data values.

```
from pprint import pprint # Import pprint library to make the out pretty.

foo = 'Lorem Ipsum' # string
health = 200        # int
mana   = 10.20      # float
lucky_numbers = [4, 13, ] # list | can be any data type | MUTABLE

body_parts    = ('head',   # Tuples | can be any data type | IMMUTABLE
                 'body',   # Immutable means that once created, it cannot
                 'legs',)  # modified after.

user = {                         # Dict | Dictionary
    'name' : 'James',            # Dictionaries are used to store data values in key:val
    'username': 'jamesp101',      # Dictionary cannot have the same key.
    'password': 'HelloWorld!',
    'age': 12,                   # The value can be of any data type
    'favcolors': ['red', 'blue'],
    'lucky_number': lucky_numbers
}

print(foo)
print(health)
print(mana)
print(lucky_numbers)
```

```
pprint(user)                    # Prettify output

Lorem Ipsum
200
10.2
[4, 13]
{'age': 12,
 'favcolors': ['red', 'blue'],
 'lucky_number': [4, 13],
 'name': 'James',
 'password': 'HelloWorld!',
 'username': 'jamesp101'}
```

2.1 Scope

The location where we can find a variable and also access it if required.

2.1.1 Global Scope

Global variables are the ones that are defined and declared outside any function. They can be used by any part of the program.

```
my_var = 12
my_str = 'Hello World'

def say_hello():
   print (my_str) # my_str variable can be called here

say_hello()

Hello World
```

2.1.2 Local Scope

Local scope variables are variables that lives only inside a block of code (e.g. function, conditionals, loops).

```
def say_hello ():
    my_str = 'Hello World'
    print (my_str)
```

```
say_hello()

Hello World
```

Variables cannot go outside the block that belongs.

```
def hello():
    my_str = 'Hello World'
    print(my_str)

print(my_str)
```

Think of the scope lifetime of a variable. Once a block of code is finished executing, the variable inside will be also removed from the memory.

```
import random

def say_hello ():
    my_str = 12

    while my_str < 100:
        random = random.randrange(1,6) # Generate random number between 1-5
        my_str += random
```

The `my_str` variable lives throughout the function. it can be accessed inside the loop. While the `random` variable cannot go outside the loop due to its scope.

3 Data Types

3.1 Built-in Data Types

Type	Keyword
Text	`str`
Numbers	`int`, `float`, `complex`
Sequence	`list`, `tuple`, `range`
Maps	`dict`
Sets	`set`, `frozenset`
Boolean	`bool`
Binary	`bytes`, `bytearray`, `memoryview`
Null	`NoneType`

4 Conditionals

Python supports the usual logical conditions from mathematics

- a == b
- a != b
- a < b
- a > b
- a <= b
- a >= b

These conditions can be used in several ways, most commonly in `if statements` and loops

```
a = 133
b = 200

if b > a:
    print("b is greater than a")

else:
    print("b is less than a")
```

```
b is greater than a
```

```
a = 133
b = 200

if b > a:
    print("b is greater than a")

elif b < a:
    print("b is less than a")

else:
    print("They are equal")
```

```
b is greater than a
```

4.1 Match Statements

NOTE: This only works with Python **3.10** or newer.

In python we don't have a switch statement. Instead we can use the `switch` statement.

```
def print_status_code (code):
    match code:
        case "200":
            print("OK")
        case "404":
            print("Not Found")
        case "500":
            print("Internal Server Error")
        case _:
            print("Invalid Status code")

print_status_code('200')
print_status_code('777')
```

```
OK
Invalid Status code
```

5 Loops

5.1 For Loops

For loop is used in iterating a list or tuples

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print (x)
```

```
apple
banana
cherry
```

Use `enumerate()` to get the number iteration.

```
fruits = ["apple", "banana", "cherry"]
for itr, x in enumerate(fruits):
    print (itr, x)
```

```
0 apple
1 banana
2 cherry
```

A for loop with range.

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

```
0
1
2
3
4
```

5.2 While Loop

With the `while` loop we can execute a set of statements as long as a condition is true.

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

```
1
2
3
4
5
```

While loop with else statement

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

else:
    print('i is no longer less than 6')
```

```
1
2
3
4
5
i is no longer less than 6
```

6 Functions

A function is a block of code that runs when it is called.

```
def my_function ():
  print("Hello from a function")
```

6.1 Calling a function

Before a function runs, it must be called.

```
def my_function ():
  print("Hello from a function")

my_function()
```

```
Hello from a function
```

Functions are useful for programmers to divide their programs into separate modules.

6.2 Return

A return statement is used to **end** the execution of the function call and return a value.

```
def my_func():
  a = 12
  b = 3
  return
a = b * a                      # This code will not be executed.
```

Note: Return statement cannot be used outside the function.
The value next to the return statement will be returned.

```
import random                    # Import random library

def roll_dice():
    return random.randrange(1,7)  # Generate a random number between 0-6 and returns it

dice_num = roll_dice()          # The returned value will be passed to the 'dice_num'var
# print(roll_dice())            # We can also print directly the function.

if dice_num == 1:
    print('You won!')
else:
    print('You lose!')
```

```
You won!
```

6.3 Parameters and Arguments

6.3.1 Parameters

Data can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

```
def my_function (fname):        # Any data that will be passed will take the 'fname' va
  print (f'My name is {fname}')

my_function('Slim shady')       # The value of 'fname' variable is 'Slim Shady'
my_function('Monad')
my_function('Giovanni Giorgio')
my_function('Antonio Montaya')

My name is Slim shady
My name is Monad
My name is Giovanni Giorgio
My name is Antonio Montaya
```

6.3.2 Arguments

The argument data will be the parameter variable.

```
def display_name (fname, lname, age,):
    print (f'My name is {fname} {lname}. I am {age} years old.')

display_name('Yoshikage', 'Kira', 33)

My name is Yoshikage Kira. I am 33 years old.
```

Arguments are the data that goes to the function call.

```
def square_nums (num):          # The 'num' is the parameter.
    return num ** 2

my_num = 12
result = square_nums(my_num)    # The 'my_num' is the argument.
print(result)


144
```

The parameters **must be filled** with data or variables, else it returns an error.

```
My name is Yoshikage Kira. I am 33 years old.
```

6.3.3 Optional Parameters / Default Value Parameters

Not all parameters are required. We can make some of the parameters **optional**.

For example the `print()` function. `print()` function provides us with multiple optional parameters. (see the docs here).

The ending character of each `print()` function is `\n`. We can change its value of it with an optional argument.

```
print('Hello World')      # No optional arguments
print('Hello', end='')    # With optional Arguments
print(' World', end='!')  # With optional Arguments
```

```
Hello World
Hello World!
```

In creating an optional parameter, it must have a default value if no arguments are provided.

```
def square_num(num=0):
    return num ** 2

print(square_num(5))
print(square_num())
```

```
25
0
```

6.4 Type Hints

In creating a function, we can indicate the **expected** data types of arguments or return values.

This can help to improve code readability, prevent errors, and make code easier to maintain

```
def add_numbers(a: int, b: float) -> float:

    return a + b


add_numbers("1", 2)  # We would get an error in the output
```

7 Exception Handling

Adding extensive error handling is crucial when developing maintainable code.

If you have a block of code that might fail, you can manage any exceptions by placing an `try:` and `except:` block.

```
import random                    # Import Random library

try:
    num = random.randrange(0, 11)    # Generate random number between 0,10
    result = 2 / num                 # If we get 0 in the 'num' variable, we wil get a
except:
    print('Cannot divide zero') # This will run if 'num' variable is zero.
```

We can specify what error we want to handle

```
try:
    num = input('Give me num: ' )
    num = float (num)
    result = 2 / num
except ValueError:               # If we cannot convert our input, this will run.
    print('Your input is wrong')
except ZeroDivisionError:
    print('Cannot divide zero') # If the input is zero
except Exception:
    print('An error occurred') # If there are other errors that occurs.


Give me num: An error occurred
```

We can have many exceptions in our try-except statement (see the built-in exceptions here).

8 Class

Python is an object-oriented programming language. Almost everything in python is an object, with its properties and methods.

A class is like an object constructor or a blueprint for creating objects.

Create a class with a property named x.

```
class MyClass:
  x=5
```

Create an object named p1, and print the value of x:

```
class MyClass:
  x=5

p1 = MyClass()
print(p1.x)
```

```
5
```

8.1 __init__() function

All classes have a function called `__init__()`, which is always executed when the class is being initiated. Use the `__init__()` function to assign values to object properties or other operations that are necessary to do when the object is being created.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person('John', 37)

print(p1.name)
print(p1.age)
```

```
John
37
```

NOTE: The ~__init__() ~function is called automatically every time the class is being used to create a new object.

8.2 Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

8.3 Self Parameter

The `self` parameter is a reference to the current instance of the class and is used to access variables that belong to the class. It does not have to be named self, you can call it whatever you like, but it has to be the first parameter of any function in the class

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

```
Hello my name is John
```

8.4 Pass Statement

`class` definitions cannot be empty, but if you for some reason have a `class` definition with no content, put in the `pass` statement to avoid getting an error.

```
class Person:
  pass
```

9 Keywords

Keywords a reserved words in python.

We cannot use a keyword as a name for a variable, function or any other identifier

We can show the reserved keywords in python by:

```
import keyword
print(keyword.kwlist)
```

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', '
```