

COL106

Introduction to Python Handout

Nilanjan Ghosh, Kavya Chopra, Tejas Kumar, Sarwagya Prasad,
Harsh Vardhan Singh, Priyansh Singh

27 July 2024

Python is a programming language with a clean and easy-to-learn syntax. It is used in web and internet development, scientific apps, desktop apps, education, and general software applications. This document would help you get started and write good apps quickly.

Contents

1 Warmup	2
1.1 The Basics	2
1.2 Input/Output	2
1.3 Code Blocks and Indentation	3
1.4 Data Types	3
1.5 Variables	4
1.6 Looping Constructs	4
1.7 Strings, Lists, and Tuples	6
1.8 Functions	9
2 Object Oriented Programming	10
2.1 Motivation	10
2.2 Classes	10
2.3 Objects	11
2.4 Encapsulation and Polymorphism	11
2.5 Fields	13
2.6 Wrappers and Decorators	13
2.7 Methods	14
3 Exceptions	15
3.1 Introduction	15
3.2 Handling Exceptions	15
3.3 Raising Exceptions	15
4 OOP Exercise Problem	16

1 Warmup

In order to run Python programs, you will need to have the Python interpreter and, possibly, a graphical editor. The Python interpreter is responsible for executing Python code, which is sometimes referred to as programs.

A program can consist of one or more Python files. These code files can also include other files or modules. When running a program, it is necessary to specify a parameter when executing Python. We will be using Python 3.7+ exclusively throughout this course. You can run a python program with the following syntax:

```
$ python3 file.py
```

1.1 The Basics

1.1.1 Identifiers

Like other languages that you may have encountered in the past, an identifier can be used to identify a variable, function, class, module, or another object. Python has some that one needs to follow while using identifiers, namely:

- An identifier starts with a letter A to Z or a to z or an underscore (.) followed by letters, digits (0 to 9) or underscores.
- The identifiers are case-sensitive.
- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is **private**.
- Starting an identifier with two leading underscores indicates a **strongly private** identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

1.2 Input/Output

1.2.1 Output

- The print() function is used to output data to STDOUT (generally, STDOUT is just the terminal where the code is ran). For example:

```
## Program to print the string "Hello World" onto the console
print("Hello World!")
```

- You can print multiple objects in the same command, and python will print them with a whitespace in between:

```
## Output: Hello, my age is 18
print("Hello, my age is", 18)
```

- By default, python adds an newline character (\n) after the print.
- The print function has the following parameters:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- The sep parameter can change the character to be inserted between the objects to be printed.
- The end parameter can change what is to be printed *after* all the objects. So, for example, if you want that your program doesn't add a newline, you can put end=""

```
## Output: Hello, my age is 18
print("Hello, my name is ", end="")
print(18)
```

1.2.2 Input

- the `input()` function is used to take input from a user, for example:

```
value = input()
```

- The input is always read as a string.
- The input string can be converted to an integer or float using the `int()` or `float()` function respectively.
- You can optionally pass a string to the input function, which will be displayed to the user before expecting an input

```
age = int(input("Enter your age: "))
```

1.3 Code Blocks and Indentation

1.3.1 Blocks of Code and Indentation

- No braces to indicate blocks of code for class and function definitions or flow control.
- Blocks of code are denoted by line indentation, which is rigidly enforced.
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

1.3.2 Statement Suites

- A group of individual statements, which make a single code block are called **suites** in Python.
- Compound or complex statements, such as `if`, `while`, `def`, and `class` require a header line and a suite.
- Header lines begin the statement (with the keyword) and terminate with a colon (`:`) and are followed by one or more lines that make up the suite.

```
if count < 10:  ## Header Line
    data = input()
print ("end suite")
```

Truth value of Objects In Python any object can be tested to be **True** or **False** for use in an `if` or `while` or Boolean operations. The following objects are considered **False**.

- **None**
- **False**
- Zero of any numeric type: 0, 0.0 and 0j.
- Any empty sequence: empty string, empty tuple, and empty list.
- Empty mappings: empty dictionary.

All other objects are considered to be **True**.

1.4 Data Types

The built in data types can be categorized in the following categories –

1. Numeric Types (integers, float and complex)
2. Sequence Types (String, List, Tuple and Range)
3. Mapping Types (Dictionary)
4. Set Types (Sets and Frozensets)

We will slowly incorporate all the above in our programs.

1.4.1 Numeric Types

There are three distinct numeric types: integers, floating point numbers, and complex numbers. In addition, booleans are a subtype of integers.

```
Integer: 48, -48, 0x260, -0x260, 0o41, -0o131, 0b111001000, -0b11100110.  
Float: 3.9, -45.6, 32.3E18, 23.19E-3  
Complex: 2j, 2.3j, 3 + 4j
```

In Python, value of a numeric type is not restricted by the number of bits and can expand to the limit of the available memory. When a binary arithmetic operator has operands of different numeric types, the operand with the narrower type is widened to that of the other, where integer is narrower than floating point, which is narrower than complex. When the type of the input numbers are same the result will be a number of the same type. Except the case of division.

There are many operators that can be used on numeric types, these are defined in Table 1. We should note here that numeric-types are immutable, every time we make a change we are also allocating a new object of the same type.

```
a = 10  
print(id(a))  
## 4351955672  
a = a + 5  
print(id(a))  
## 4351955832
```

We will be covering sequence, mapping, and sets types later in detail after variables and loops.

1.5 Variables

Like any programming language, python has variables to store, reference or manipulate data. The identifier used to define these variables refers to an instance of a class.

```
a = int(input("Enter a number: "))  
print(a+1)
```

- In Python, we do not need to declare variables (which is different from languages like C and C++). A variable is declared the first time it is assigned a value.
- Python is a dynamically typed language. This means that the same variable can be used to store different data types at different points of the program.

```
##Output:  
## Datatype of a: <class 'int'>  
## Datatype of a: <class 'str'>  
a = 18  
print("Datatype of a:", type(a))  
a = "Hello"  
print("Datatype of a:", type(a))
```

If you further investigate these variables with the `id()` function, you'll find that the identifier has changed, and the same variable is now pointing to a different object.

1.6 Looping Constructs

1.6.1 Range

Objects of type range are created using the `range()` function. This is a versatile function to create iterables yielding arithmetic progressions.

```
range(start, stop, step)
```

- You can see how this syntax can be used in Table 2. We will be using these range functions to loop through iterables and loops.

Table 1: Some operators that can be used with numeric types

Operator	Operation	Usage
+	Addition	a+b
-	Subtraction	a-b
*	Multiplication	a*b
/	Division	a/b
%	Modulus. Divides left-hand operand by right-hand operand and returns the remainder	a%b
**	Exponent	a ** b
//	Floor division. Divides left hand operand with right hand operand, rounds off the result to the closest but lower integer	9//4 will yield 2 and -9/4 will yield -3.
==	Returns True if both operands are equal.	a == b
!=	Returns True if both operands are not equal.	a != b
>	Returns true if left operand is more than right operand.	a>b
<	Return true if right operand is more than left operand.	a=	Return true of left operand I more than or equal to right operand.	a>=b
<=	Returns true if left operand is less than or equal to right operand.	a <= b
is	Object comparison. Evaluates to true if the names on either side of the operator point to the same object and false otherwise.	x is y essentially checks the identity (as returned by the id() function) of the two objects – x and y.
is not	Opposite of above.	Opposite of above.
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+=	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a , The same can be done for other arithmetic operators.
&	Binary AND: Operator copies a bit to the result if it exists in both operands	a & b = 12 (0b0000 1100)
	Binary OR: It copies a bit if it exists in either operand.	(a b) = 61 (0b0011 1101)
^	Binary XOR: It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (0b0011 0001)
~	Binary Ones Complement: It is unary and has the effect of 'flipping' bits.	(~a) = -61 (-0b11110100)
<<	Binary Left Shift: The left operand value is moved left by the number of bits specified by the right operand.	a <<2 = 240 (0b1111 0000)
>>	Binary RightShift: The left operands value is moved right by the number of bits specified by the right operand.	a >>2 = 15 (0b0000 1111)
and	Logical AND: If both operands are true then condition becomes true.	(a and b) is False.
or	Logical OR: If any of the two operands are true, then the condition becomes true.	(a or b) is True.
not	Logical Not: Used to reverse the logical state of its operand.	not (a and b) is True.

Table 2: Using range to generate iterables

Syntax	Meaning
<code>range(4, 16, 3)</code>	Generates the sequence: 4, 7, 10, 13
<code>range(4, 16, -3)</code>	Makes no sense. Will not generate anything
<code>range(16, 4, -3)</code>	Generates the sequence: 16, 13, 10, 7
<code>range(4, 16, -3)</code>	Makes no sense. Will not generate anything.
<code>range(10)</code>	Generates 0, 1, 2, ... 8, 9
<code>range(-5)</code>	Makes no sense. Will not generate anything

1.6.2 While Loop Construct

- A while loop in Python looks a lot similar to C-like languages and Java. The loop continues till the condition is met.

```
counter = 0
while (counter < 7):
    print(counter)
    counter = counter + 1
```

1.6.3 For Loop Construct

- The for statement iterates over the items of any sequence (list, string, range, among others) in the order that they appear in the sequence.

```
for counter in range(0,8,1):
    print(counter)
```

Exercise 1

Write a program that takes two numbers, A and B. Find numbers between A and B where each digit is an even number. How many such numbers are there?

Exercise 2

Write a program that accepts a positive number and subtracts the sum of its digits from this number. Repeat this process until the number becomes negative or zero, and print the number of such operations required to make the resulting number non-positive.

1.7 Strings, Lists, and Tuples

1.7.1 Strings

- Strings are a special type of sequence that can only store characters.
- Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.
- Python does not have a character data type, a single character is simply a string with a length of one.
- You can embed single quote within a double quoted string and double quote within a single quoted string. Backward slash can be used to escape quotes. Triple quotes can be used for strings spanning multiple lines.

```
string = "Hello World!"
str1 = str(12)
str2 = ''' This is a multi-line
string statement'''
```

- Strings are immutable (once created they cannot be modified). Concatenating or splitting a string would create a new string object.

Table 3: Functions Specific to Strings

Function	Explanation
<code>replace(old, new, [count])</code>	Replace count occurrences of the old string with the new string. If the count is not specified, replace all occurrences.
<code>title()</code> , <code>capital()</code> , <code>lower()</code> <code>islower()</code> , <code>isupper()</code>	Returns a new string in title, capital, and lower cases, respectively. The last two methods return <code>True</code> if all the characters in the string are lower and smaller, respectively.
<code>swapcase()</code>	Returns another string with all uppercase characters converted to lowercase and vice versa of the given string.
<code>isalpha()</code> , <code>isalnum()</code> , <code>isdigit()</code>	Returns <code>True</code> if the string has only alphabets (<code>abc..zABC..Z</code>). The second function returns <code>True</code> if the string has only alphabets (<code>abc..zABC..Z</code>) and/or digits (<code>12..0</code>) without a decimal point. The third function returns <code>True</code> if the string has digits only without a decimal point.
<code>strip([str])</code> , <code>lstrip([str])</code> , <code>rstrip([str])</code>	Strip <code>str</code> from both the ends, the leading end or the trailing end, respectively. If no argument is provided, whitespaces (including tabs and newlines) are stripped.

```
string[3] = "0"
## Traceback (most recent call last):
##   File "<stdin>", line 1, in <module>
## TypeError: 'str' object does not support item assignment
```

- There are some functions that are only applicable for strings. Some of these are laid out in Table 3.

Exercise 3

Write a program to replace all subsequent occurrences of the first character of a given string with another given character (#).

- Sample input: jumping jack.
- Expected output: jumping #ack.

1.7.2 Lists

- List can be written as a list of comma-separated values (items) between square brackets.
- Important thing about the list is that items in a list need not be of the same type.
- Lists are mutable (they can be changed).

```
nums = [1,2,3,4,5]
mix = [1,"name",98.4]
nums[3] = 8
print(nums)
## [1, 2, 3, 8, 5]
```

- While we can make rudimentary changes like this we can also loop through the list.

```
for i in nums:
    mix.append(i*2)
print(mix)
## [1, 'name', 98.4, 2, 4, 6, 16, 10]
```

Exercise 4

Write a program to get the sum of comma-separated numbers. If the sum is more than 100, print 0. Else, print the actual sum.

Table 4: Functions Specific to Lists

Function	Explanation
<code>list.append(x)</code>	Add an item to the end of the list. Equivalent to <code>a[len(a):] = [x]</code>
<code>list.extend(iterable)</code>	Extend the list by appending all the items from the iterable. Equivalent to <code>a[len(a):] = iterable</code> .
<code>list.insert(i, x)</code>	Insert an item at a given position. The first argument is the index of the element before which to insert
<code>list.remove(x)</code>	Deletes the x from the list. If x is not found it throws <code>ValueError</code> .
<code>list.pop(i)</code>	Returns the ith item and deletes it from the sequence.
<code>list.clear()</code>	Remove all items from the list. Equivalent to <code>del a[:]</code> .
<code>list.index(x)</code>	Return the zero-based index in the list of the first item whose value is equal to x. Raises a <code>ValueError</code> if there is no such item.
<code>list.count(x)</code>	Return the number of times x appears in the list.
<code>list.sort(key=None, reverse=False)</code>	Sort the items of the list in place (the arguments can be used for sort customization, see <code>sorted()</code> for their explanation).
<code>list.reverse()</code>	Reverse the elements of the list in place.
<code>list.copy()</code>	Return a shallow copy of the list. Equivalent to <code>a[:]</code> .

1.7.3 Tuples

- A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The major differences between tuples and lists are - tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets. Tuples are constructed by the comma operator (not within square brackets), with or without enclosing parentheses.
- Operations with tuples are much faster than operations with lists. The performance difference can be partially measured using the `timeit` library which allows you to time your Python code. The code below, which simply creates a tuple and a list consisting of exactly the same elements, is run 1 million times and its total time of execution is measured using the `timeit` library.

```
import timeit
timeit.timeit('x=(1,2,3,4,5,6,7,8,9,10,11,12)', number=1000000)
timeit.timeit('x=[1,2,3,4,5,6,7,8,9,10,11,12]', number=1000000)
## 0.02018076300737448 0.1307151880027959
```

1.7.4 Operations Applicable to All Sequences

- **Inclusion Check:** The operator `in` returns `True` if the given item is part of the given sequence and vice-versa. The operator `not in` returns `True` if the given item is not part of the given sequence and vice-versa. If this is applied over a string this operation is equivalent to substring test.

```
string = 'healthy wealthy and wise'
mylist = ['wise', 48, 29.3]
substring = 'health'
print (substring in string)
print (substring not in string)
print (substring in mylist)
## True
## False
## False
```

- **Concatenation:** The operators `+` and `*` concatenates given sequences. The asterisk operator creates a shallow copies of the sequence and appends them to the sequence. Concatenating immutable sequences always results in a new object.
 - A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
 - A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.
- **Element Access:** Using square brackets and 0 based indexing individual elements can be accessed.

```
tuple1 = ('health', 48, 23.9, 'wealth')
print(tuple1[2])
## 23.9
```

Please look at the following concepts as they may serve a better pythonic way of writing solutions: object slicing, object unpacking, list comprehension.

1.8 Functions

To group sets of code you can use functions. Functions are small parts of repeatable code. A function accepts parameters and follows this syntax while defining one:

```
def f(x,y):
    return x,y,x+y
print(f(48,3))
## (248, 3, 51)
print(f("hello","world"))
## ('hello', 'world', 'helloworld')
```

Please note the following points, x and y are variables with the scope of the function. These do not have a type defined and thus can be used for adding as well as concatenating. Further, we return three values. They are returned in the form of a tuple which can be unpacked or stored in a variable.

Exercise 5

Given a list of integers with duplicate elements in it. The task is to generate another list that contains only duplicate elements. The new list should include elements that appear more than once.

Example 1:

- Input: 10, 20, 30, 20, 20, 30, 40, 50, -20, 60, 60, -20, -20

- Output: [20, 30, -20, 60]

Example 2:

- Input: -1, 1, -1, 8

- Output: -1

2 Object Oriented Programming

2.1 Motivation

Object-oriented programming (OOPs) is a programming technique that makes use of objects and classes to create a program in a way that is both intuitive and efficient. The main motivation behind OOPs is to model real world entities as software objects that have both state (attributes) and behavior (methods).

Imagine you're a wizard trying to manage a world filled with magical creatures. Without OOPs, you'd have to handle each creature's attributes and behaviors individually, leading to chaos and a lot of repetitive code. With OOPs, you can create a 'Creature' class and then create instances of different creatures (objects) that inherit the attributes and behaviors of the 'Creature' class.

2.2 Classes

A class is like a blueprint for creating objects. It defines the attributes and methods that the objects created from the class will have. Let's create a class for our magical creatures.

```
class Creature:
    # Class attribute
    kingdom = "Magicland"

    def __init__(self, name, species):
        # Instance attributes
        self.name = name
        self.species = species

    def introduce(self):
        # Method to introduce the creature
        print(f"Hi, I'm {self.name} the {self.species} from {self.kingdom}."
              )

    def perform_magic(self):
        # Method to perform magic
        print(f"{self.name} performs a spectacular magical act!")
```

2.2.1 Data and Methods in Class Definition

In the 'Creature' class:

- **Class Attributes:** These are attributes shared by all instances of the class. In our example, `kingdom` is a class attribute with the value "Magicland".
- **Instance Attributes:** These are attributes specific to each instance of the class. In our example, `name` and `species` are instance attributes.
- **Methods:** Functions defined within a class that describe the behaviors of the objects. In our example, `introduce` and `perform_magic` are methods. More about methods is described later in this handout.

2.2.2 Constructors and Destructors

Constructors and Destructors

- **Constructor (`__init__` method):** A special method that is called when an object is instantiated. It initializes the object's attributes. This method allows you to set up any initial state or perform setup tasks necessary for the object.
- **Destructor (`__del__` method):** A special method that is called when an object is about to be destroyed. It can be used to clean up resources or perform any necessary finalization. However, in Python, the destructor is less commonly used due to automatic garbage collection, which handles most of the cleanup for you.

Now, let's create some creatures using our enhanced 'Creature' class:

```

# Creating objects (instances of the class)
unicorn = Creature("Sparkle", "Unicorn")
dragon = Creature("Blaze", "Dragon")

# Calling methods on the objects
unicorn.introduce() # Hi, I'm Sparkle the Unicorn from Magicland.
dragon.introduce() # Hi, I'm Blaze the Dragon from Magicland.
unicorn.perform_magic() # Sparkle performs a spectacular magical act!
dragon.perform_magic() # Blaze performs a spectacular magical act!

# When the objects are deleted or go out of scope, the destructor is called
del unicorn
del dragon

```

With our ‘Creature’ class, we can easily create and manage multiple magical creatures. Each creature has its own `name` and `species`, but they all share the same `kingdom`. The constructor initializes these attributes when a new creature is created, and the destructor cleans up when the creature is no longer needed.

By using OOPs, we can make our code more modular, reusable, and easier to understand.

2.3 Objects

Objects are instances of classes. When a class is defined, no memory is allocated until an object of that class is created. Objects are the fundamental building blocks in object-oriented programming and represent real-world entities with attributes (state) and behaviors (methods).

```

# Creating objects (instances of the class)
unicorn = Creature("Sparkle", "Unicorn")
dragon = Creature("Blaze", "Dragon")

```

Each object has its own copy of the instance attributes defined in the class. Objects can interact with each other and can be passed as arguments to functions.

2.4 Encapsulation and Polymorphism

2.4.1 Encapsulation

Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, i.e., a class. It restricts direct access to some of an object’s components, which is a means of preventing accidental interference and misuse of the data.

In Python, encapsulation is achieved using private and protected access modifiers:

- **Private Attributes/Methods:** Prefixing an attribute or method with double underscores (e.g., `__private_attribute`) makes it private, meaning it cannot be accessed directly from outside the class.
- **Protected Attributes/Methods:** Prefixing an attribute or method with a single underscore (e.g., `_protected_attribute`) makes it protected, indicating that it should not be accessed from outside the class, but is still accessible.

Differences Between Private and Protected

- **Private:** Private members are accessible only within the class that defines them. They are not accessible from outside the class, not even by subclasses.
- **Protected:** Protected members are accessible within the class and by subclasses. However, they are not intended to be accessed directly from outside the class hierarchy.

Why Use Private and Protected Methods?

- **Private Methods:** Used to hide the internal implementation details and protect the object’s integrity by preventing external code from accessing or modifying internal state directly.
- **Protected Methods:** Used to allow access to methods and attributes within the class and its subclasses, promoting code reuse while still controlling access.

Use Case

Consider a bank account system where you need to ensure that the account balance cannot be modified directly except through specific methods like deposit or withdraw.

```
class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.__account_number = account_number # Private attribute
        self.__balance = initial_balance # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            self.__update_account_summary() # Private method

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            self.__update_account_summary() # Private method

    def get_balance(self):
        return self.__balance # Getter method for balance

    def _update_account_summary(self): # Protected method
        # Protected method to update account summary
        pass

    def __update_account_summary(self):
        # Private method to update account summary
        print(f"Account {self.__account_number}: New balance is {self.
            __balance}")

# Example usage
account = BankAccount("123456789", 1000)
account.deposit(500)
print(account.get_balance())
account.withdraw(200)
print(account.get_balance())
# Direct access to private attribute will raise an error
# account.__balance = 10000 # This will raise an AttributeError
```

In this example, `__balance` is a private attribute, ensuring it cannot be modified directly. Methods like `deposit` and `withdraw` provide controlled ways to modify the balance. The private method `__update_account_summary` ensures internal consistency whenever the balance is updated.

2.4.2 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It is the ability of different objects to respond to the same message (or method call) in different ways. This is typically achieved through method overriding.

Example of Polymorphism with Shapes

```
class Shape:
    def draw(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Circle(Shape):
    def draw(self):
        return "Drawing a circle"

class Square(Shape):
    def draw(self):
```

```

        return "Drawing a square"

def draw_shape(shapes):
    for shape in shapes:
        print(shape.draw())

# Example usage
shapes = [Circle(), Square()]
draw_shape(shapes) # Outputs: Drawing a circle\nDrawing a square

```

2.5 Fields

Fields, also known as attributes, are variables that belong to a class and define the state or properties of an object.

- **Class Attributes:** Shared across all instances of the class. They are defined within the class but outside any instance methods.
- **Instance Attributes:** Specific to each instance of the class. They are typically defined within the constructor (`__init__` method).

```

class Creature:
    kingdom = "Magicland" # Class attribute

    def __init__(self, name, species):
        self.name = name # Instance attribute
        self.species = species # Instance attribute

```

2.6 Wrappers and Decorators

Wrappers and decorators are powerful tools in Python that allow you to modify the behavior of functions or methods.

2.6.1 Wrappers

A wrapper is a function that is used to extend the behavior of another function. This is often used to add some kind of pre- or post-processing to the original function.

```

def wrapper_function(original_function):
    def new_function(*args, **kwargs):
        # Pre-processing
        print("Something is happening before the function is called.")
        result = original_function(*args, **kwargs)
        # Post-processing
        print("Something is happening after the function is called.")
        return result
    return new_function

@wrapper_function
def say_hello():
    print("Hello!")

```

2.6.2 Decorators

A decorator is a special type of wrapper that is applied to functions or methods using the `@decorator_name` syntax. Decorators provide a clear and readable way to modify the behavior of functions or methods.

```

def decorator_function(original_function):
    def wrapper_function(*args, **kwargs):
        print("Wrapper executed this before {}".format(original_function.
            __name__))
        return original_function(*args, **kwargs)

```

```

    return wrapper_function

@decorator_function
def display():
    print("Display function ran")

display()

```

In this example, `decorator_function` is a decorator that wraps the `display` function. When `display` is called, the wrapper function is executed before and after the original function.

Decorators can also be used to apply the same modification to multiple functions, making your code more DRY (Don't Repeat Yourself).

2.7 Methods

Methods are functions defined within a class that describe the behaviors of the objects created from the class. There are different types of methods in Python:

- **Instance Methods:** Operate on an instance of the class. They can access and modify the object's attributes. Defined using `def` and the first parameter is always `self`.
- **Class Methods:** Operate on the class itself rather than on instances of the class. Defined using `@classmethod` decorator and the first parameter is always `cls`.
- **Static Methods:** Do not operate on the instance or the class. Defined using `@staticmethod` decorator and do not take `self` or `cls` as the first parameter.

```

class Creature:
    kingdom = "Magicland"

    def __init__(self, name, species):
        self.name = name
        self.species = species

    def introduce(self): # Instance method
        print(f"Hi, I'm {self.name} the {self.species} from {self.kingdom}.")
        )

    @classmethod
    def describe_kingdom(cls): # Class method
        print(f"This is the kingdom of {cls.kingdom}.")

    @staticmethod
    def is_magic_creature(species): # Static method
        return species in ["Unicorn", "Dragon", "Phoenix"]

```

- **Instance Methods:** Called on an instance of the class and can access or modify instance attributes.
- **Class Methods:** Called on the class itself and can modify class attributes. They use the `@classmethod` decorator.
- **Static Methods:** Independent of class and instance attributes. They use the `@staticmethod` decorator.

3 Exceptions

3.1 Introduction

In Python, exceptions are a way to handle errors or unusual conditions that occur during the execution of a program. When an error occurs, Python generates an exception, which is an object that describes the error. If not handled, exceptions can cause a program to terminate abruptly.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

3.2 Handling Exceptions

Fortunately, Python provides us with a mechanism to handle these exceptions, and take appropriate actions where needed. This is done with the help of a `try-except` block.

```
while True:
    try:
        x = int(input("Please enter an integer value "))
        break
    except ValueError:
        print("You have entered an invalid value, try again!")
```

A `ValueError` is encountered when `int(X)` fails on a non-number like string input `X`. This piece of code prompts the user to enter an integer, until a valid integer value is entered.

You can also handle multiple different types of exceptions in the same `try-except` block. For instance,

```
try:
    with open('example.txt', 'r') as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("The file was not found.")
except PermissionError:
    print("You do not have permission to access the file.")
except IsADirectoryError:
    print("Expected a file but found a directory.")
except IOError as e:
    print(f"An I/O error occurred: {e}")
```

3.3 Raising Exceptions

You can raise exceptions too, to signify invalid input, or if the program control flow results in something you explicitly want to forbid. For example:

```
def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero!")
    return x / y
```

4 OOP Exercise Problem

Problem 6

Develop a class structure using Python to manage courses, students, and faculty within a college setting to demonstrate the practical utility of object-oriented programming.

Here's a description of the classes and methods you're supposed to implement:

- **CollegeMember**: Base class representing any member within the college ecosystem.
 - *Attributes*:
 - * **name** (string): The full name of the college member.
 - * **member_id** (string): A unique identifier for the college member.
 - *Methods*:
 - * **display_info()**: Prints the name and ID of the college member. No input parameters.
- **Student**: Inherits from *CollegeMember*.
 - *Additional attributes*:
 - * **courses** (list of strings): A list storing the names of courses the student is currently enrolled in.
 - *Methods*:
 - * **enroll(course: string)**: Adds a new course to the student's list of courses. The method checks if the course is not already enrolled before adding to the list to prevent duplication.
 - * **drop_course(course: string)**: Removes a course from the student's list of courses.
 - * **display_info()**: Overridden to also print a list of courses the student is currently enrolled in, alongside the name and ID.
- **Faculty**: Inherits from *CollegeMember*.
 - *Additional attributes*:
 - * **courses_teaching** (list of strings): A list storing the names of courses the faculty member is currently teaching.
 - * **password** (string): A private attribute used to protect sensitive information.
 - *Methods*:
 - * **assign_teaching(course: string)**: Assigns a new course for the faculty member to teach. The method checks if the course is not already being taught before adding to the list.
 - * **remove_course(course: string)**: Removes a course from the faculty member's list of courses being taught.
 - * **display_info(password: string)**: Overridden to require a password. If the password matches the stored password, it prints the faculty's name, ID, and courses they are teaching. Otherwise, it raises an exception.

Implementation Code: Please run the following Python code to check the correct implementation of your classes and their methods:

```
# [IMPLEMENT YOUR CLASSES HERE]
# Asserting inheritance
assert issubclass(Student, CollegeMember), "Student should inherit from
CollegeMember"
assert issubclass(Faculty, CollegeMember), "Faculty should inherit from
CollegeMember"

# Create instances of students and faculty
member = CollegeMember("Sarwagy Prasad", "Mem001")
student1 = Student("Kavya Chopra", "S001")
student2 = Student("Tejas Kumar", "S002")
faculty1 = Faculty("Amit Kumar", "amitk", "12345678")
faculty2 = Faculty("Parag Singla", "parags", "password")

# Enrolling students in courses
student1.enroll("Introduction to Python")
student1.enroll("Data Structures")
student2.enroll("Machine Learning")
student2.enroll("Software Engineering")
```

```

# Assigning courses to faculty
faculty1.assign_teaching("Introduction to Python")
faculty2.assign_teaching("Machine Learning")
faculty1.assign_teaching("Data Structures")
faculty2.assign_teaching("Software Engineering")

# Displaying initial information with correct and incorrect passwords
print("Initial Information:")
member.display_info()
student1.display_info()
student2.display_info()
try:
    faculty1.display_info("12345678") # Correct password
except:
    print("Password Error")
try:
    faculty2.display_info("wrongpassword") # Incorrect password
except:
    print("Password Error")

# Updating course enrollments and assignments
student1.drop_course("Data Structures")
student1.enroll("Advanced Python Techniques")
student2.enroll("Machine Learning") # check for duplication
faculty1.remove_course("Introduction to Python")
faculty1.assign_teaching("Advanced data structures and Algorithms")

# Displaying updated information with password checks
print("Updated Information:")
student1.display_info()
student2.display_info()
try:
    faculty1.display_info("12345678")
except:
    print("Password Error")

## TRY PRINTING NAME AND PASSWORD OF A FACULTY WITHOUT CALLING DISPLAY_INFO
print(faculty1.name)
# try print faculty1 password similarly

## EXPECTED : NAME SHOULD BE PRINTED WHEREAS PASSWORD MUST NOT BE

```