National University of Computer and Emerging Sciences

Lab Manual 03
Introduction to Data Science

Instructor: Rida Amir

Objectives

1. Functions - Learn how to define, call, and use functions.
2. Classes and Instances - Understand how to create objects (like real things) using classes and instances in programming.
3. Exception handling - Gain the ability to detect and manage errors or exceptions in code to ensure robust program execution.
4. File handling - Learn how to open, read, and save data in files using your program.

A. Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- In Python a function is defined using the def keyword. To call a function, use the function name followed by parenthesis:

Example

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

my_function()
```

- Information 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.
- The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

- Python also has a set of built-in functions. Refer to this link to see a list. https://www.w3schools.com/python/python_ref_functions.asp

B. Classes and Instances

- A **class** is a blueprint or template for creating objects.
- It defines **attributes** (properties) and **methods** (functions) that the objects created from the class will have.

Creating a Class

- Use the keyword `class` followed by the class name (usually starting with a capital letter).
- Inside the class, define a special method called `__init__` which initializes the object's attributes.
- The `__init__` method takes `self` as the first parameter (which represents the instance itself) and other parameters for attributes.

Defining Attributes

- Attributes are variables that hold data about the object.
- Inside `__init__`, assign values to attributes using `self.attribute_name = value`.
- Attributes describe the object's characteristics.

Defining Methods

- Methods are functions defined inside a class that describe behaviors or actions.
- They always have `self` as the first parameter to access the instance's attributes and other methods.
- Methods can perform operations using the object's data or modify the object's state.

Creating Instances (Objects)

- An **instance** is an actual object created from the class blueprint.
- To create an instance, call the class name with arguments for the attributes.
- Each instance has its own copy of the attributes.

Example: Creating a Car Class and Using Instances in Python

```python
class Car:
    def __init__(self, make, model, year):
        self.make = make      # Attribute for car make
        self.model = model    # Attribute for car model
        self.year = year      # Attribute for car manufacturing year

    def display_info(self):
        # Method to print car details
        print(f"{self.year} {self.make} {self.model}")
```

2. **Create Instances:**

```python
car1 = Car("Toyota", "Corolla", 2020)
car2 = Car("Honda", "Civic", 2018)
```

---

3. **Access Attributes and Call Methods:**

```python
car1.display_info()  # Output: 2020 Toyota Corolla
car2.display_info()  # Output: 2018 Honda Civic
```

4. **Modify Attributes:**

```python
car2.model = "Accord"  # Change the model of car2
car2.display_info()    # Output: 2018 Honda Accord
```

Reading material : https://www.w3schools.com/python/python_classes.asp

C. Python Exception Handling

An exception is an error that is thrown by our code when the execution of the code results in an unexpected outcome. Normally, an exception will have an error type and an error message. Some examples are as follows.

```
ZeroDivisionError: division by zero
TypeError: must be str, not int
```

ZeroDivisionError and TypeError are the error type and the text that comes after the colon in the error message. The error message usually describes the error type.

Types of Exceptions

Here's a list of the common exceptions you'll come across in Python:

1. **ImportError**: It is raised when you try to import the library that is not installed or you have provided the wrong name
2. **IndexError:** Raised when an index is not found in a sequence. For example, if the length of the list is 10 and you are trying to access the 11th index from that list, then you will get this error
3. **IndentationError**: Raised when indentation is not specified properly
4. **ZeroDivisionError**: It is raised when you try to divide a number by zero
5. **ValueError**: Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified
6. **Exception**: Base class for all exceptions. If you are not sure about which exception may occur, you can use the base class. It will handle all of them

Exception Handling with Try Except Clause

Python provides us with the try except clause to handle exceptions that might be raised by our code. The basic anatomy of the try except clause is as follows:

```
try:
   // some code
except:
   // what to do when the code in try raise an exception
```

In plain English, the try except clause is basically saying, "Try to do this, except (otherwise) if there's an error, then do this instead".

There are a few options on what to do with the thrown exception from the `try` block. Let's discuss them.

Catch certain types of exception

Another option is to define which exception types we want to catch specifically. To do this, we need to add the exception type to the `except` block.

```
try:
    myfunction(100, "one hundred")
except TypeError:
    print("Cannot sum the variables. Please pass numbers only.")


print("This WILL be printed")
```

To make it even better, we can actually log or print the exception itself.

```
try:
    myfunction(100, "one hundred")
except TypeError as e:
    print("Cannot sum the variables. The exception was:" ,e)

#Cannot sum the variables. The exception was: unsupported operand type(s) for +: 'int' and 'str'
```

Furthermore, we can catch multiple exception types in one `except` clause if we want to handle those exception types the same way. Let's pass an undefined variable to our function so that it will raise the NameError. We will also modify our `except` block to catch both `TypeError` and `NameError` and process either exception type the same way.

```
try:
    myfunction(100, a)
except (TypeError, NameError) as e:
    print("Cannot sum the variables. The exception was" , e)

#Cannot sum the variables. The exception was name 'a' is not defined
```

Try….Finally

So far the try statement had always been paired with except clauses. But there is another way to use it as well. The try statement can be followed by a **finally** clause. Finally clauses are called clean-up or termination clauses, because they must be executed under all circumstances, i.e. a

"finally" clause is always executed regardless if an exception occurred in a try block or not. A simple example to demonstrate the finally clause:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
finally:
    print("There may or may not have been an exception.")
print("The inverse: ", inverse)
```

Your number: 34
There may or may not have been an exception.
The inverse: 0.029411764705882353

Try..except and finally

"finally" and "except" can be used together for the same try block, as it can be seen in the following Python example:

```
try:
    x = float(input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print("You should have given either an int or a float")
except ZeroDivisionError:
    print("Infinity")
finally:
    print("There may or may not have been an exception.")
```

D. File handling

Opening a File

- Use the built-in open() function to open a file.
- Syntax: open(filename, mode)
- Modes include:
    - 'r' — read (default mode)
    - 'w' — write (creates a new file or overwrites existing)

- 'a' — append (adds data at the end of the file)
- 'r+' — read and write

Reading from a File

- Use methods like:
  - read() — reads entire file content as a string
  - readline() — reads one line at a time
  - readlines() — reads all lines into a list

Writing to a File

- Use write() method to write strings to the file.
- Writing will overwrite existing content in 'w' mode or add content in 'a' mode.

Closing a File

- Always close the file after opening using close() to free resources.
- Or better, use a with statement to open files, which automatically closes the file when done.

Operation	Method	Description
Open file	`open()`	Open file with mode
Read entire file	`file.read()`	Read full content
Read one line	`file.readline()`	Read next line
Read all lines	`file.readlines()`	Read lines as list
Write to file	`file.write()`	Write string to file
Close file	`file.close()`	Close the file

Example: Reading and Writing a File

```python
# Writing to a file
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("Welcome to Python file handling.\n")

# Reading from the file
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

Output:

```css
Hello, World!
Welcome to Python file handling.
```