AI LAB # 1

Online Python Compiler (https://www.programiz.com/python-programming/online-compiler/)

LAB 1 Introduction to Python – I

1.1 Essentials of a Python program

- 1.1.1 Flow of control
- 1.1.2 Indentation and (lack of) semicolons
- 1.1.3 Built-in types
- 1.1.4 Files
- 1.1.5 Variable scope and lifetime

1.2 Selection control statements

- 1.2.1 Selection: if statement
- 1.2.2 The for Statement

1.3 Defining Functions

- 1.3.1 Default Argument Values
- 1.3.2 Lambdas

1.4 Class Definitions:

- 1.4.1 The self-Identifier
- 1.4.2 The Constructor
- 1.4.3 Class and Instance Variables

1.5 Lab Tasks

Introduction to Python

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991. It is used for:

- web development
- software development,
- mathematics,

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

1.1 Essentials of Python

In most of today's written languages, words by themselves do not make sense unless they are in certain order and surrounded by correct punctuation symbols. This is also the case with the Python programming language. The Python interpreter is able to interpret and run correctly structured Python programs. For example, the following Python code is correctly structured and will run:

```
print("Hello, world!")
```

Many other languages require a lot more structure in their simplest programs, but in Python this single line, which prints a short message, is sufficient. A very informative example of Python's syntax which does (almost) exactly the same thing:

```
# Here is the main function.
def my_function():
    print("Hello, World!")

my_function()
```

A hash (#) denotes the start of a comment. The interpreter will ignore everything that follows the hash until the end of the line.

1.1.1 Flow of control

In Python, statements are written as a list, in the way that a person would write a list of things to do. The computer starts off by following the first instruction, then the next, in the order that they appear in the program. It only stops executing the program after the last instruction is completed. We refer to the order in which the computer executes instructions as the flow of control. When the computer is executing a particular instruction, we can say that control is at that instruction.

1.1.2 Indentation and (lack of) semicolons

Many languages arrange code into blocks using curly braces **({**and}**)** or BEGIN and END statements – these languages encourage us to indent blocks to make code easier to read, but indentation is not compulsory. Python uses indentation only to delimit blocks, so we must indent our code:

```
# this function definition starts a new block
def add_numbers(a, b):
    # this instruction is inside the block, because it's indented
    c = a + b
    # so is this one
    return c

# this if statement starts a new block
if it_is_tuesday:
    # this is inside the block
    print("It's Tuesday!")
# this is outside the block!
print("Print this no matter what.")
```

In many languages we need to use a special character to mark the end of each instruction – usually a semicolon. Python uses ends of lines to determine where instructions end.

```
# These all individual instructions -- no semicolons required!
print("Hello!")
print("Here's a new instruction")
a = 2

# This instruction spans more than one line
b = [1, 2, 3,
    4, 5, 6]

# This is legal, but we shouldn't do it
c = 1; d = 5
```

1.1.3 Built-in types

In programming, data type is an important concept. Variables can store data of different types, and different types can do different things. Python has the following data types built-in by default, in these categories:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview

Getting the Data Type

You can get the data type of any object by using the type () function:

```
x = 5
print (type(x))
```

Setting the Data Type

In Python, the data type is set when you assign a value to a variable:

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict

x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray
x = memoryview(bytes(5))	memoryview

Numbers

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

Type Conversion

You can convert from one type to another with the int(), float(), and complex() methods:

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex

#convert from int to float:
a = float(x)
```

```
#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

Note: You cannot convert complex numbers into another number type.

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages.

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

Strings

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

```
print("Hello")
print('Hello')
```

Assign String to a Variable

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

```
a = "Hello"
print(a)
```

Multiline Strings

You can assign a multiline string to a variable by using three quotes:

Example

You can use three double quotes:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Or three single quotes:

```
a = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(a)
```

Note: in the result, the line breaks are inserted at the same position as in the code.

Strings are Arrays

Like many other popular programming languages, strings in Python are arrays of bytes representing Unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

```
a = "Hello, World!"
print(a[1])
```

String Length

To get the length of a string, use the len() function.

```
a = "Hello, World!"
print(len(a))
```

Some common escape sequences:

Sequence	Meaning
\\	literal backslash
\'	single quote
\"	double quote
\n	newline
\t	tab

Sometimes we may need to define string literals which contain many backslashes – escaping all of them can be tedious. We can avoid this by using Python's raw string notation. By adding an **r** before the opening quote of the string, we indicate that the contents of the string are exactly what we have written, and that backslashes have no special meaning. For example:

```
# This string ends in a newline
"Hello!\n"

# This string ends in a backslash followed by an 'n'
r"Hello!\n"
```

Consider another example:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'ununununium'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
'tho'
```

1.1.4 Files

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

demofile.txt

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in open() function.

The open() function returns a file object, which has a read() method for reading the content of the file:

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

If the file is located in a different location, you will have to specify the file path, like this:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Read Only Parts of the File

By default the read() method returns the whole text, but you can also specify how many characters you want to return. Return the 5 characters of the file.

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read Lines

You can return one line by using the readline() method:

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

By calling readline() two times, you can read the two first lines:

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

Close Files

It is a good practice to always close the file when you are done with it.

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

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Example

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

1.1.5 Variable scope and lifetime

Where a variable is accessible and how long it exists depend on how it is defined. We call the part of a program where a variable is accessible its scope, and the duration for which the variable exists its lifetime. A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file, and also inside any file which imports that file. Global variables can have unintended consequences because of their wide-ranging effects – that is why we should almost never use them. Only objects which are intended to be used globally, like functions and classes, should be put in the global namespace. A variable which is defined inside a function is local to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing.

Here is an example of variables in different scopes:

```
# This is a global variable
a = 0

if a == 0:
    # This is still a global variable
    b = 1

def my_function(c):
    # this is a local variable
    d = 3
    print(c)
    print(d)

# Now we call the function, passing the value 7 as the first and only parameter
my_function(7)

# a and b still exist
print(a)
print(b)

# c and d don't exist anymore -- these statements will give us name errors!
print(c)
print(d)
```

1.2 Selection control statements

1.2.1 Selection: if statement

Perhaps the most well-known statement type is the if statement. For example:

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more **elif** parts, and the else part is optional. The keyword '**elif**' is short for '**else if**', and is useful to avoid excessive indentation. An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

The interpreter will treat all the statements inside the indented block as one statement – it will process all the instruction in the block before moving on to the next instruction. This allows us to specify multiple instructions to be executed when the condition is met.

1.2.2 The for Statement

The **for** statement in Python differs a bit from what you may be used to in C. Rather than always giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example:

```
for i in range(1, 9):
    print(i)
```

Consider another example.

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

If you do need to iterate over a sequence of numbers, the built-in function **range ()** comes in handy. It generates arithmetic progressions:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; range(10) generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
range(5, 10)
   5, 6, 7, 8, 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

1.3 Defining Functions

A function is a sequence of statements which performs some kind of task. Here is a definition of a simple function which takes no parameters and doesn't return any values:

```
def print_a_message():
    print("Hello, world!")
```

We use the **def** statement to indicate the start of a function definition. The next part of the definition is the function name, in this case print_a_message, followed by round brackets (the definitions of any parameters that the function takes will go in between them) and a colon. Thereafter, everything that is indented by one level is the body of the function.

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

1.3.1 Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: ask_ok('Do you really want to quit?')
- giving one of the optional arguments: ask_ok('OK to overwrite the file?', 2)
- or even giving all arguments: ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

This example also introduces the **in** keyword. This tests whether or not a sequence contains a certain value.

1.3.2 Lambdas

We have already seen that when we want to use a number or a string in our program we can either write it as a literal in the place where we want to use it or use a variable that we have already defined in our code. For example, print ("Hello!") prints the literal string "Hello!", which we haven't stored in a variable anywhere, but print(message) prints whatever string is stored in the variable message.

We have also seen that we can store a function in a variable, just like any other object, by referring to it by its name (but not calling it). Is there such a thing as a function literal? Can we define a function on the fly when we want to pass it as a parameter or assign it to a variable, just like we did with the string "Hello!"?

The answer is yes, but only for very simple functions. We can use the lambda keyword to define anonymous, one-line functions inline in our code:

```
a = lambda: 3

# is the same as

def a():
    return 3
```

Lambdas can take parameters – they are written between the lambda keyword and the colon, without brackets. A lambda function may only contain a single expression, and the result of evaluating this expression is implicitly returned from the function (we don't use the return keyword):

```
b = lambda x, y: x + y

# is the same as

def b(x, y):
    return x + y
```

1.4 Class Definitions:

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

To create a class, use the keyword class:

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

Create Object

Now we can use the class named MyClass to create objects:

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

1.4.1 The Self Identifier

Self represents the instance of the class. By using the "self" keyword we can access the attributes and methods of the class in python. It binds the attributes with the given arguments.

The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically: the first parameter of methods is the instance the method is called on.In more clear way you can say that SELF has following Characteristic-

Self is always pointing to Current Object.

```
# It is clearly seen that self and obj is referring to the same object
class check:

    def _init_(self):

        print("Address of self = ",id(self))

obj = check()
```

```
print("Address of class object = ",id(obj))
```

Output

```
Address of self = 140124194801032
Address of class object = 140124194801032
```

Another Example of Using SELF:

```
class car():
    # init method or constructor
    def _init_(self, model, color):

        self.model = model

        self.color = color

    def show(self):

        print("Model is", self.model )

        print("color is", self.color )


# both objects have different self which
# contain their attributes

audi = car("audi a4", "blue")

ferrari = car("ferrari 488", "green")


audi.show()     # same output as car.show(audi)

ferrari.show()  # same output as car.show(ferrari)
```

Self is the first argument to be passed in Constructor and Instance Method.

Self must be provided as a First parameter to the Instance method and constructor. If you don't provide it, it will cause an error.

1.4.2 The Constructor

Constructors are generally used for instantiating an object. The task of constructors is to initialize (assign values) to the data members of the class when an object of the class is created. In Python the _init_ () method is called the constructor and is always called when an object is created.

Syntax of constructor declaration:

def _init_(self):

    # body of the constructor

Types of constructors:

Default constructor: The default constructor is a simple constructor which doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed.

Parameterized constructor: constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

1.4.3 Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

1.5 Lab Task:

Task 1: Find the type of following statements. Is that value an integer or a floating point or string?

a. `250`

b. `89i`

c. `Lab`

Task 2: Consider the following output as shown:

```
*
* *
* * *
* * * *
* * * * *
```

Task 3: Consider the following output as shown:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Task 4: Consider the following output as shown:

```
A
B B
C C C
D D D D
E E E E E
```