AI320 - Artificial Intelligence

Spring Semester 2022

Instructor: Dr. Diaa Salama Abdel Moneim
Eng. Haytham Metawie
Eng. Mostafa Badr

FEBRUARY 16, 2022

Lab 1 – Introduction to Python - Part 1

Topic	Duration
Introduction to Python - Part 1	2 hours

Document Revision History

Revision Number	File Name	Date
1	Lab 1 – Introduction to Python - Part 1	16/2/2022

I. Getting started with python:

➢ Why do people use python?

First of all, we must admit that the choice of development tools is sometimes based on **unique constraints** or **personal preference**.

But there are some primary factors in which python excels, such as:

- **Developer productivity:**

  Python boosts developer productivity many times beyond compiled or statically typed languages such as C, C++, and Java. Python code is typically one-third to one-fifth the size of equivalent C++ or Java code.

- **Program portability:**

  Most Python programs run unchanged on all major computer platforms. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script's code between machines.

- **Support libraries:**

  Python comes with a large collection of prebuilt and portable functionality, known as the standard library. This library supports an array of application-level programming tasks, from text pattern matching to network scripting.

  Python can be extended with both homegrown libraries and a vast collection of third-party application support software. Python's third-party domain offers tools for website construction, numeric programming, serial port access, game development, and much more.

  The NumPy extension, for instance, has been described as a free and more powerful equivalent to the MATLAB numeric programming system.

- **Component integration:**

  Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms.

  Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components and can interact over networks.

- **Enjoyment:**

  Because of Python's ease of use and built-in toolset, it can make the act of programming more pleasure than chore.

➢ How Python runs programs?

Python is not just a programming language; it's also a software package called an **interpreter**.

An **interpreter** is a kind of program that executes other programs. When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains.

The **interpreter** itself may be implemented as a C program, a set of Java classes, or something else.

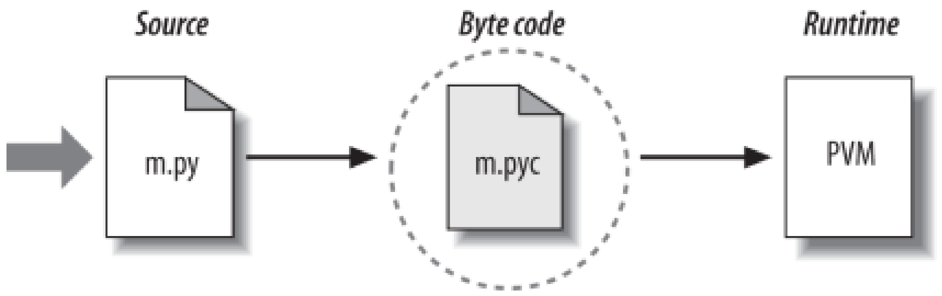
The interpretation process is composed of 2 steps:

1. **Compilation into byte code**

   Compilation is simply a translation step, and byte code is a lower-level, platform-independent representation of your source code.

   Python interpreter translates each of your source statements into a group of byte code instructions by decomposing them into individual steps.

   This byte code translation is performed to speed execution, byte code can be run much more quickly than the original source code statements in your text file.



   Python saves byte code as a startup speed optimization. The next time you run your program, Python will load the .pyc files (compiled .py files) and skip the compilation step, as long as you haven't changed your source code since the byte code was last saved, and aren't running with a different Python than the one that created the byte code.

   Finally, keep in mind that byte code is saved in files only for files that are imported, not for the top-level files of a program that are only run as scripts.

   Moreover, a given file is only imported (and possibly compiled) once per program run, and byte code is also never saved for code typed at the **interactive prompt**.

2. **Running on the Python Virtual Machine (PVM)**

Once your program has been compiled to byte code (or the byte code has been loaded from existing .pyc files), it is shipped off for execution to something generally known as the **Python Virtual Machine**.

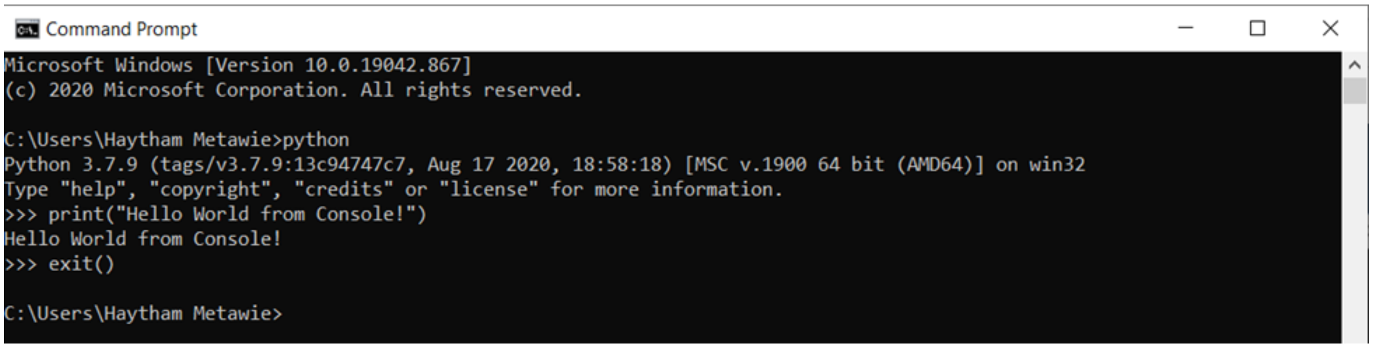
PVM is not a separate program, and it need not be installed by itself. In fact, the PVM is just a big code loop that iterates through your byte code instructions, one by one, to carry out their operations.

The PVM is the runtime engine of Python; it's always present as part of the Python system, and it's the component that truly runs your scripts.

Now, which is faster Python code or C++ code? And why?

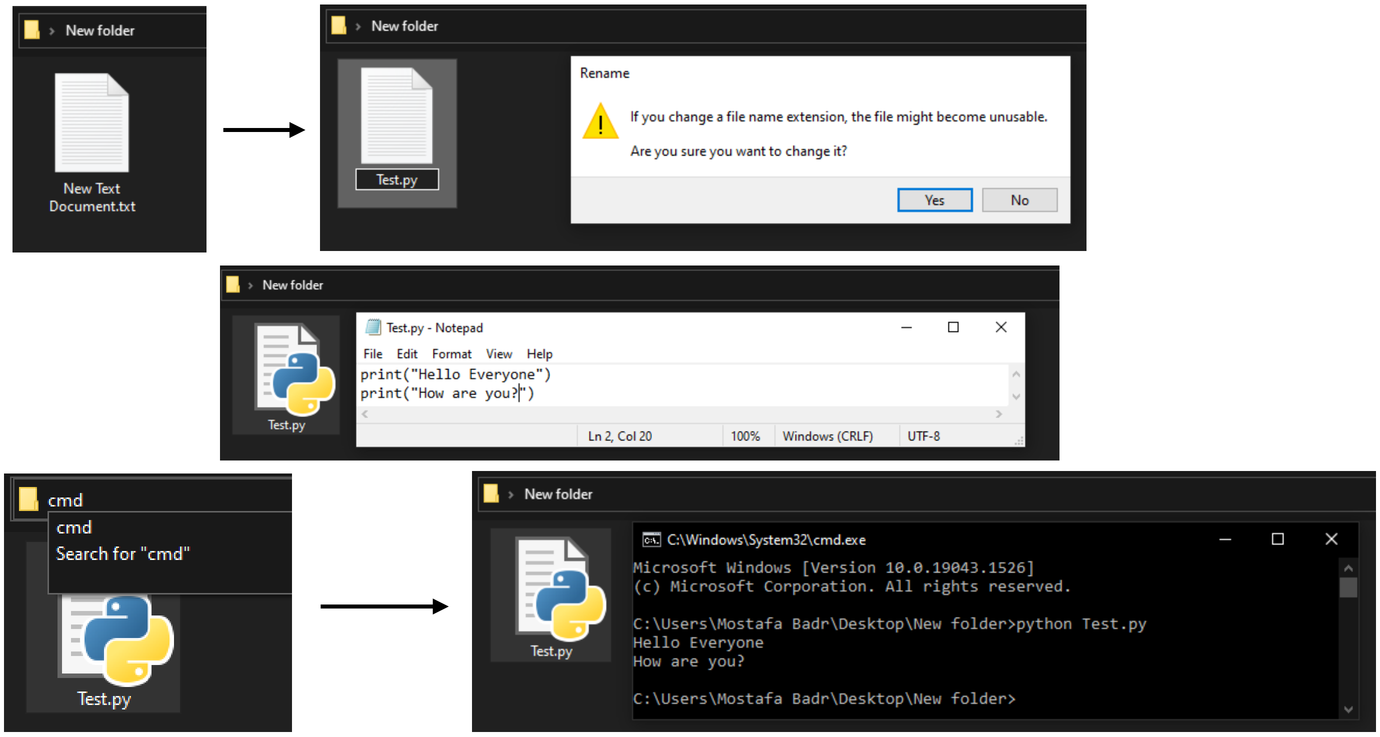
➢ How you run programs?

- You can run python code directly using console (cmd) (Interactive prompt).



```
Microsoft Windows [Version 10.0.19042.867]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\Users\Haytham Metawie>python
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 18:58:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World from Console!")
Hello World from Console!
>>> exit()

C:\Users\Haytham Metawie>
```

- You can also run python scripts (files) using console (cmd).



If you change a file name extension, the file might become unusable.

Are you sure you want to change it?

```
print("Hello Everyone")
print("How are you?")
```

```
Microsoft Windows [Version 10.0.19043.1526]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Mostafa Badr\Desktop\New folder>python Test.py
Hello Everyone
How are you?

C:\Users\Mostafa Badr\Desktop\New folder>
```

- We can also use IDEs for coding and debugging (which will be discussed later).

➢ General Notes on Python

To end a statement in Python, you do not have to type in a semicolon or other special character, you simply press enter. Semicolons can be used to separate statements if you wish to put multiple statements in the same line.

Comments in Python can be done in two manners: single line comment is done by beginning the statement with hash mark (#) or block comment which is done by placing a block of code/statements in between of three successive single quotes (‘’’) (or double quotes).

To print out a message, you do not need to include/import any library. Place what you want to inside parentheses preceded by print as depicted in previous figures.

Indentation matters in Python. As you will see later in the manual may results in irrational behavior of your program or even errors.

II. Python types and operations:

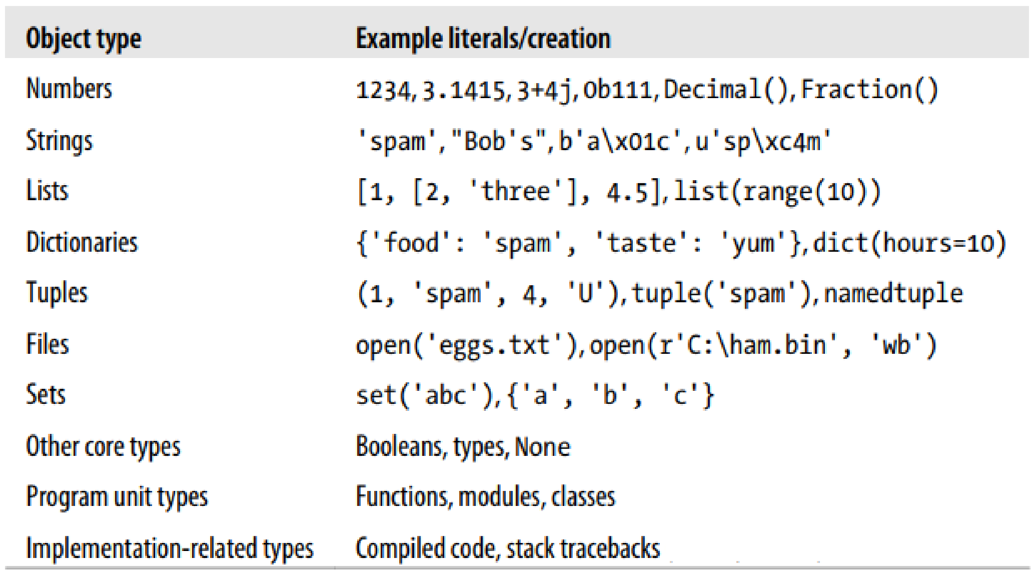
Now let’s begin our tour of the Python language.

In Python, data takes the form of objects, either **built-in objects** that Python provides, or **objects we create using Python classes**.

Objects are essentially just pieces of memory, with values and sets of associated operations. (Everything is an object in a Python script)

There are **no type declarations** in Python, the syntax of the expressions you run determines the types of objects you create and use.

➢ Primitive (Built-in) data types

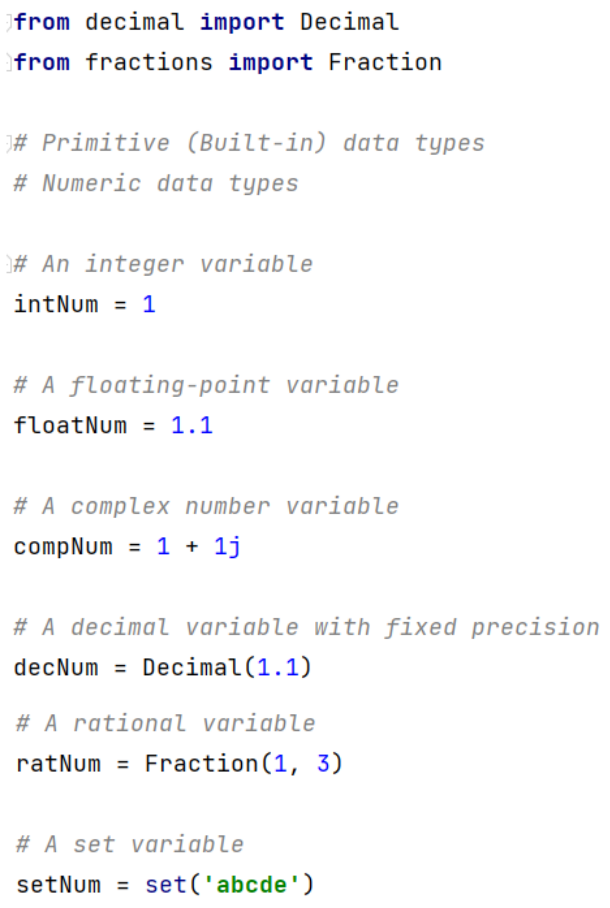


Object type	Example literals/creation
Numbers	`1234`, `3.1415`, `3+4j`, `0b111`, `Decimal()`, `Fraction()`
Strings	`'spam'`, `"Bob's"`, `b'a\x01c'`, `u'sp\xc4m'`
Lists	`[1, [2, 'three'], 4.5]`, `list(range(10))`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}`, `dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U')`, `tuple('spam')`, `namedtuple`
Files	`open('eggs.txt')`, `open(r'C:\ham.bin', 'wb')`
Sets	`set('abc')`, `{'a', 'b', 'c'}`
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes
Implementation-related types	Compiled code, stack tracebacks

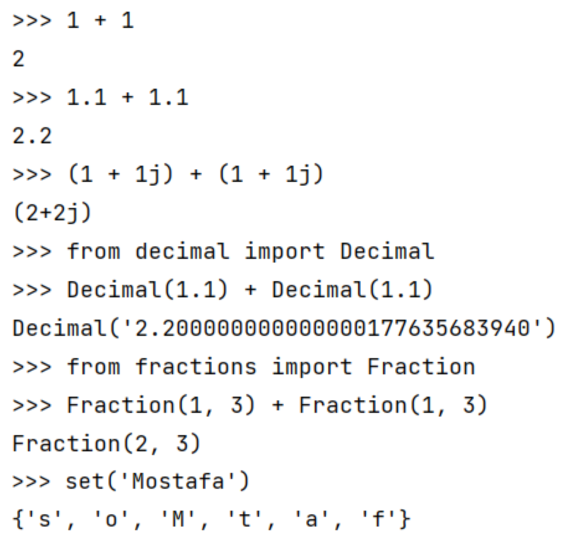
Now let's dive through some of these built-in data types

1. Numeric data types

Python's core objects set includes the usual suspects: **integers** that have no fractional part, **floating-point** numbers that do, and more exotic types, **complex numbers** with imaginary parts, **decimals** with fixed precision, **rational** with numerator and denominator, and **full-featured sets**.



```python
from decimal import Decimal
from fractions import Fraction

# Primitive (Built-in) data types
# Numeric data types

# An integer variable
intNum = 1

# A floating-point variable
floatNum = 1.1

# A complex number variable
compNum = 1 + 1j

# A decimal variable with fixed precision
decNum = Decimal(1.1)

# A rational variable
ratNum = Fraction(1, 3)

# A set variable
setNum = set('abcde')
```



```
>>> 1 + 1
2
>>> 1.1 + 1.1
2.2
>>> (1 + 1j) + (1 + 1j)
(2+2j)
>>> from decimal import Decimal
>>> Decimal(1.1) + Decimal(1.1)
Decimal('2.200000000000000177635683940')
>>> from fractions import Fraction
>>> Fraction(1, 3) + Fraction(1, 3)
Fraction(2, 3)
>>> set('Mostafa')
{'s', 'o', 'M', 't', 'a', 'f'}
```

2. Textual data types (strings)

Strings are used to **record both textual information** (your name, for instance) as well as **arbitrary collections of bytes** (such as an image file's contents). They are our first example of what in Python we call a **sequence** which is a positionally ordered collection of other objects.

Strings are sequences of one-character strings; other, more general sequence types include lists and tuples.

```
# Textual variable (string)
myName = "Haytham Metawie"
myFavNum = '7'
emptyString = ""
```

```
>>> myName = "Haytham Metawie"
>>> myFavNum = '7'
>>> myName + "'s favourite number is " + myFavNum
"Haytham Metawie's favourite number is 7"
```

In Python, indexes are coded as offsets from the front, and so start from 0, we can also index backward (from the end), positive indexes count from the left, and negative indexes count back from the right.

```
>>> myName = "Mostafa"
>>> myName[0]
'M'
>>> myName[-1]
'a'
```

In addition to simple positional indexing, sequences also support a more general form of indexing known as **slicing**, which is a way to extract an entire section (slice) in a single step.

```
>>> sentence = "I have an Idea"
>>> sentence[2:6]
'have'
```

In a slice, the left bound defaults to zero, and the right bound defaults to the length of the sequence being sliced. (HINT: Extended sclicing)

```
>>> sentence[2:]
'have an Idea'
>>> sentence[:6]
'I have'
>>> sentence[:]
'I have an Idea'
```

Python also supports repetition in a simple way

```
>>> sentence * 3
'I have an IdeaI have an IdeaI have an Idea'
```

Every string operation is defined to produce a new string as its result, because strings are **immutable** in Python, they cannot be changed in place after they are created.

you can never overwrite the values of immutable objects. For example, you can't change a string by assigning to one of its positions, but you can always build a new one and assign it to the same name.

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

Every object in Python is classified as either immutable (unchangeable) or not. In terms of the core types, **numbers**, **strings**, and **tuples** are immutable; **lists**, **dictionaries**, and **sets** are not, they can be changed in place freely, as can most new objects you'll code with classes.

Python provides methods and call patterns for built-in string objects; the following table is for Python version 3.3.

```
S.capitalize()
S.casefold()
S.center(width [, fill])
S.count(sub [, start [, end]])
S.encode([encoding [,errors]])
S.endswith(suffix [, start [, end]])
S.expandtabs([tabsize])
S.find(sub [, start [, end]])
S.format(fmtstr, *args, **kwargs)
S.index(sub [, start [, end]])
S.isalnum()
S.isalpha()
S.isdecimal()
S.isdigit()
S.isidentifier()
S.islower()
S.isnumeric()
S.isprintable()
S.isspace()
S.istitle()
S.isupper()
S.join(iterable)
S.ljust(width [, fill])
S.lower()
S.lstrip([chars])
S.maketrans(x[, y[, z]])
S.partition(sep)
S.replace(old, new [, count])
S.rfind(sub [,start [,end]])
S.rindex(sub [, start [, end]])
S.rjust(width [, fill])
S.rpartition(sep)
S.rsplit([sep[, maxsplit]])
S.rstrip([chars])
S.split([sep [,maxsplit]])
S.splitlines([keepends])
S.startswith(prefix [, start [, end]])
S.strip([chars])
S.swapcase()
S.title()
S.translate(map)
S.upper()
S.zfill(width)
```

3. Boolean data type

As in all programming languages, Python supports Boolean variables.

```
# Boolean variable
true = True
false = False
```

But keep in mind two things you **MUST** assign variable with True/False not 1/0 or it will be considered storing integer values. Another thing is true and false are not data types.

4. Binary data type

Binary variables are not supported in several languages, but Python does support them. There are three different types of binary objects: **bytes**, **bytearray** and **memoryview**.

- **bytes**

  bytes object is a sequence of small integers, each of which is in the range 0 through 255, that happens to print as ASCII characters when displayed.

```
# byte variable
binString = b'python'
emptyBytes = bytes(4)
encodedBytes1 = bytes('python', 'utf8')
encodedBytes2 = bytes(b'\x01\x02\x03')
```

  bytes support indexing, slicing and most methods available for strings.

```
>>> binString = b'python'
>>> binString[1]
121
>>> binString[2:4]
b'th'
>>> binString.split(b't')
[b'py', b'hon']
```

  Note, bytes are immutable (can't be changed)

```
>>> binString[1] = b'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

- **bytearray**

  A mutable sequence of integers in the range 0 through 255, which is a mutable variant of bytes. As such, it supports the same string methods and sequence operations as bytes, as well as many of the mutable in-place-change operations supported by lists.

```
# bytearray variable
binString = b'python'
binArray = bytearray(binString)
emptyBinArray = bytearray(10)
```

  bytearray require an integer for index assignments (not a string), we use the 'ord' function to get the integer value corresponding to the string.

```
>>> binString = b'python'
>>> binArray = bytearray(binString)
>>> binArray[1] = ord('x')
>>> binArray
bytearray(b'pxthon')
```

- **memoryview**

```
# Memoryview
binString = b'python'
binMemView = memoryview(binString)
```

  Refer to this link in order to get a deeper explanation of memoryview() function.

➢ Variable Naming Restrictions

1. **Variable names in python can be any length.**

```
>>> myFullNameSinceBeenBornAndAlwaysWillBeForeverAndIWillNotChangeItEver = """Badr"""
```

2. **Variable names can contain letters (A → Z, a → z), digits (0 → 9) and underscore (_). Although variable names can contain digits, it can't start with a digit.**

```
>>> myName_1 = "Haytham"
>>> _myName_1 = "Haytham"
>>> 1_myName_1 = "Haytham"
  File "<stdin>", line 1
    1_myName_1 = "Haytham"
     ^
SyntaxError: invalid decimal literal
```

3. **Variable names can contain upper and/or lower cases letters.**

```
>>> myName1 = "Haytham"
```

4. **Reserved words cannot be used in variable names.**

```
>>> class = "b"
  File "<stdin>", line 1
    class = "b"
          ^
SyntaxError: invalid syntax
```

5. **Python is a case-senstive language which means name = “Haytham” differs from Name = “Haytham”.**

```
>>> name = "Haytham"
>>> Name = "Haytham"
```

Retrieving Objects type and ID

Python has inbuilt function to retrieve the data type of an object like **type()** and its unique ID **id()**. The id is the object's memory address and will be different for each time you run the program.

```
>>> intNum = 7
>>> print(type(intNum))
<class 'int'>
>>> print(id(intNum))
140714445187040
```

Type Casting

You can convert integer, float and string to each other as follows:

```
# Type Casting
print(str(2j))
print(float(1))
print(bool(9))
print(int("129"))
print(float("32.2"))
print(int(float("32.2")))
```

```
2j
1.0
True
129
32.2
32
```

Arithmetic and Logical Operations

Arithmetic and logical operations don’t vary much from other programming languages.

1. **Arithmetic Operations**

Operator	Description	Statement	Expected output
+	Addition	2 + 3	5
-	Subtraction	3 – 2	1
*	Multiplication	2*3	6
/	Division	3/2	1.5
%	Modulus	3%2	1
**	Exponentiation	2**3	8
//	Floor Division	3//2	1

2. Bitwise Operations

Operator	Description	Statement	Expected output
&	ADD	2 & 3	2
|	OR	2 | 3	3
~	NOT	~2	1
^	XOR	1^2	3
>>	N bits Right Shift	40 >> 3	5
<<	N bits Left Shift	3 << 2	12

3. Comparison Operations

Operator	Description
==	Equal
!=	Not Equal
<	Less than
>	Greater than
<=	Less than or Equal
>=	Greater than or Equal

4. Logical Operations

Operator	Statement	Expected output
and	2 < 3 and 3 > 2	True
or	2 < 3 or 3 < 2	True
not	not (2 < 3)	False

5. Membership Operations

Operator	Statement	Expected output
in	‘H’ in ‘Haytham’	True
not in	‘M’ not in ‘Haytham’	True

6. Identity Operations

Operator	Statement	Expected output
is	2 is 2	True
is not	‘M’ is not ‘M’	False

III. Python statements and syntax:

➢ Control flow

Python statements used for controlling the flow of your program are:

1. If, If … Else and If … Elif … Else Statements

Python's goals are to make programmers' lives easier by requiring less typing, which can be easily observed when comparing the syntax of "If" statements in both C++ and Python.

- Simple If

```python
# If, If … Else and If … Elif … Else Statements
name1 = "Haytham"
name2 = "Mostafa"
if len(name1) > len(name2):
      print("The length of name1 is greater")
print("This is always printed")
```

```
This is always printed
```

- If … Else

```python
# If, If … Else and If … Elif … Else Statements
name1 = "Haytham"
name2 = "Mostafa"
if len(name1) > len(name2):
      print("The length of name1 is greater")
else:
      print("The length of name1 is not greater")
print("This is always printed")
```

```
The length of name1 is not greater
This is always printed
```

- If … Elif … Else

```python
name1 = "Haytham"
name2 = "Mostafa"
if len(name1) > len(name2):
      print("The length of name1 is greater")
elif len(name1) == len(name2):
      print("The lengths are equal")
else:
      print("The length of name1 is not greater")
print("This is always printed")
```

```
The lengths are equal
This is always printed
```

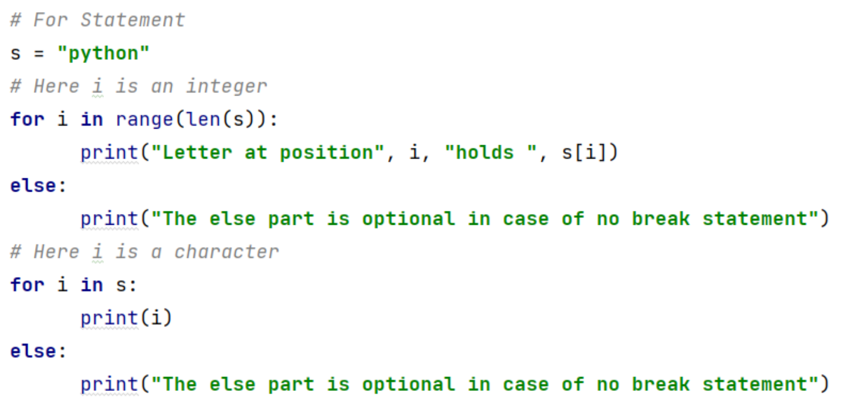
Indentation is essential in all python statements and can cause hardly noticed errors if not well treated.

How to write a nested If?

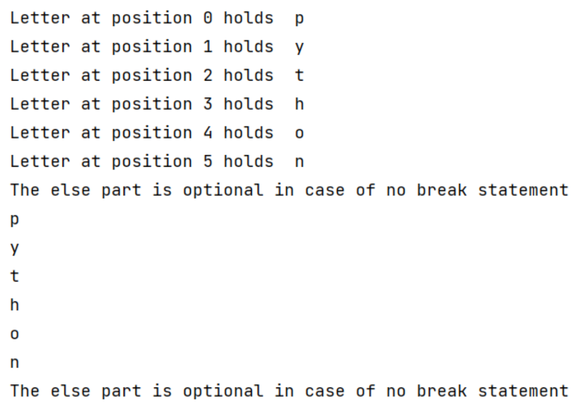
2. For Statement

As in almost all languages for loop in Python has iterator and provided number of loops but has a different syntax surely.

In Python, iterator is not limited to integer values. It can be any object such as character, string, or even user-defined object. Another merit, You do not need to declare it.



```python
# For Statement
s = "python"
# Here i is an integer
for i in range(len(s)):
      print("Letter at position", i, "holds ", s[i])
else:
      print("The else part is optional in case of no break statement")
# Here i is a character
for i in s:
      print(i)
else:
      print("The else part is optional in case of no break statement")
```

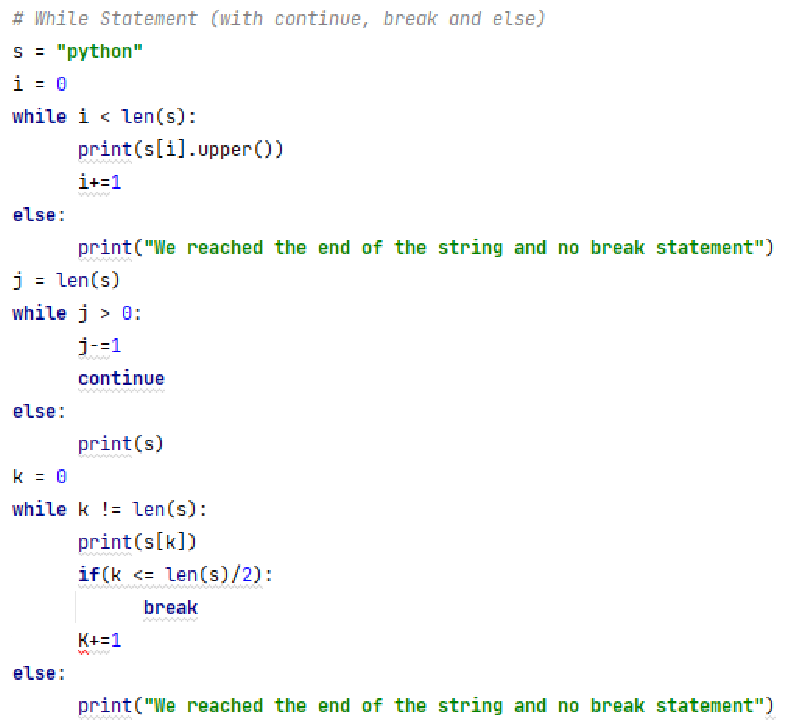


```
Letter at position 0 holds  p
Letter at position 1 holds  y
Letter at position 2 holds  t
Letter at position 3 holds  h
Letter at position 4 holds  o
Letter at position 5 holds  n
The else part is optional in case of no break statement
p
y
t
h
o
n
The else part is optional in case of no break statement
```

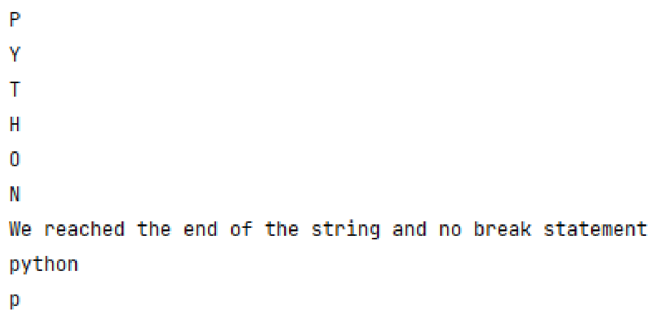
How to write a nested for loop?

3. While Statement

While Loop does not differ in Python from other languages except you may see else associated with a while loop and it is not an error.



```python
# While Statement (with continue, break and else)
s = "python"
i = 0
while i < len(s):
      print(s[i].upper())
      i+=1
else:
      print("We reached the end of the string and no break statement")
j = len(s)
while j > 0:
      j-=1
      continue
else:
      print(s)
k = 0
while k != len(s):
      print(s[k])
      if(k <= len(s)/2):
            break
      K+=1
else:
      print("We reached the end of the string and no break statement")
```



```
P
Y
T
H
O
N
We reached the end of the string and no break statement
python
p
```

Exercises:

1. **Write a program to prompt the user for hours and rate per hour to compute gross pay. Consider that the factory gives the employee 1.5 times the hourly rate for hours worked above 40 hours.**

   Expected Input/Output script:
   ```
   Enter the total worked hours:
   57
   Enter the rate per hour:
   2
   The gross pay is 131
   ```

2. **Pig Latin is a language game, where you move the first letter of the word to the end and add "ay." So "Python" becomes "ythonpay." To write a Pig Latin translator in Python, here are the steps we will need to take:**
   - Ask the user to input a word in English.
   - Make sure the user entered a valid word.
   - Convert the word from English to Pig Latin.
   - Display the translation result.

3. **Write a Python program to construct the following pattern, using a nested for loop.**
   ```
   *
   * *
   * * *
   * * * *
   * * * * *
   * * * *
   * * *
   * *
   *
   ```

4. **Write a program which prints all the divisors of a number.**

   Expected Input/Output script:
   ```
   Enter number: 21
   The divisors are 3, 7
   ```

5. **Write a program which prompts user for a number and prints the reverse of it.**
   Expected Input/Output script:
   ```
   Enter number: 834
   The reversed number is: 438
   ```

Thank you