First Python Program

Let us execute the programs in different modes of programming.

1. Interactive Mode Programming

Type the following text at the Python prompt and press Enter –

```
>>> print ("Salam, Python!")
```

By default, Python interpreter starts in interactive mode with a clean workspace.

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this if statement:

```
>>>
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

2. Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

Let us write a simple Python program in a script. Python files have the extension **.py**. Type the following source code in a test.py file –

Note: you can use any Editor for the typing such as Notepad, Sublime etc

```
print ("salam, Python!")
```

Comments in Python

Comments in Python start with the hash character, `#`, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or

code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
# This is the first comment
Spam = 1  # and this is the second comment
          # ... and now a third!
Text = "# This is not a comment because it's inside quotes."
```

Multi-line comments

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line.

Another way of doing this is to use triple quotes, either `'''` or `"""`.

These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well

```
"""This is also a
perfect example of
multi-line comments"""
```

Python Statement

Instructions that a Python interpreter can execute are called statements. For example, `a = 1` is an assignment statement. `if` statement, `for` statement, `while` statement etc. are other kinds of statements which will be discussed later.

Multi-line statement

In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
```

```
    7 + 8 + 9
```

This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as

```
a = (1 + 2 + 3 +
    4 + 5 + 6 +
    7 + 8 + 9)
```

Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

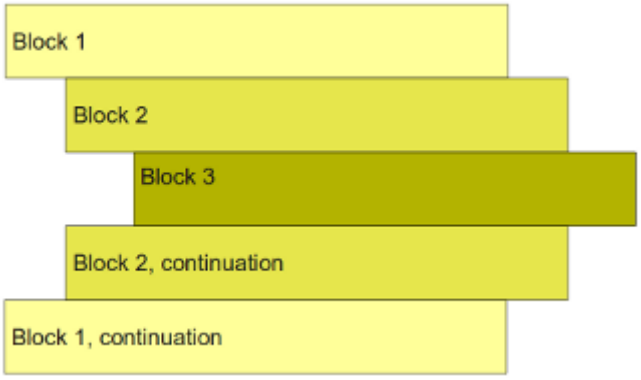
```
cars = ['parodua',
          'toyota',
          'proton']
```

We could also put multiple statements in a single line using semicolons, as follows

```
a = 1; b = 2; c = 3
```

Code Blocks and Indentation

One of the most distinctive features of Python is its use of indentation to mark blocks of code. Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.



All statements with the same distance to the right belong to the same block of code, i.e. the statements within a block line up vertically. The block ends

at a line less indented or the end of the file. If a block has to be more deeply nested, it is simply indented further to the right

Consider the if-statement from this simple password-checking program:

```
if pwd == 'ECIE3101':
    print('Logging on ...')
else:
    print('Incorrect password.')
print('All done!')
```

The lines `print('Logging on ...')` and `print('Incorrect password.')` are two separate *code blocks*. These ones happen to be only a single line long, but Python lets you write code blocks consisting of any number of statements.
To indicate a block of code in Python, you must indent each line of the block by the same amount. The two blocks of code in our example if-statement are both indented four spaces, which is a typical amount of indentation for Python.

In most other programming languages, indentation is used only to help make the code look pretty. But in Python, it is required for indicating what block of code a statement belongs to. For instance, the final `print('All done!')` is *not* indented, and so is *not* part of the else-block.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constants or variables or any other identifier names. All the Python keywords contain lowercase letters only.

and	exec	Not
as	finally	or
assert	for	pass
break	from	print
class	global	raise
continue	if	return
def	import	try
del	in	while
elif	is	with
else	lambda	yield
except		

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Keywords cannot be used as identifiers.

```
>>> global = 1
  File "<interactive input>", line 1
    global = 1
           ^
SyntaxError: invalid syntax
```

Python does not allow punctuation characters such as @, $, and % within identifiers.

```
>>> J@ = 0
  File "<interactive input>", line 1
    J@ = 0
     ^
SyntaxError: invalid syntax
```

Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers –

- 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 strong private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Python Input, Output and Import

Python provides numerous built-in functions that are readily available to us at the Python prompt. Some of the functions like `input()` and `print()` are widely used for standard input and output operations respectively

Python Output Using print() function

We use the `print()` function to output data to the standard output device (screen). We can also output data to a file

```
print('This sentence is output to the screen')
# Output: This sentence is output to the screen

a = 5
print('The value of a is', a)
# Output: The value of a is 5
```

Output formatting

Sometimes we would like to format our output to make it look attractive. This can be done by using the `str.format()` method. This method is visible to any string object.

```
>>> x = 7; y = 15
>>> print('The value of x is {} and y is {}'.format(x,y))
The value of x is 7 and y is 15
```

Here the curly braces {} are used as placeholders. We can specify the order in which it is printed by using numbers (tuple index).

```
print('I love {0} and {1}'.format('Rice','Chicken'))
# Output: I love bread and butter

print('I love {1} and {0}'.format('Chicken','Rice'))
# Output: I love butter and bread
```

Python Input Using input() function

Python allow flexibility of accepting the input from the user. In Python, we have the `input()` function to allow this. The syntax for `input()` is

```
input([prompt])
```

where `prompt` is the string we wish to display on the screen. It is optional.

```
>>> nname = input('Enter your name: ')
Enter a name: Hidayah
>>> nname
Hidayah
```

Python import

When program grows bigger, it is a good idea to break it into different modules. Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the `import` keyword to do this.

For example, we can import the `math` module by typing in `import math`.

```
import math
print(math.pi)
```

```
pi
```

```
3.141592653589793
```

Using Python as a Calculator

Numbers

Start the interpreter and wait for the primary prompt, `>>>`.

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 (for example, Pascal or C); parentheses (`()`) can be used for grouping. For example:

```
>>>
```

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

The integer numbers (e.g. `2`, `4`, `20`) have type `int`, the ones with a fractional part (e.g. `5.0`, `1.6`) have type `float`. We will see more about numeric types later in the tutorial.

Division (`/`) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the `//` operator; to calculate the remainder you can use `%`:

```
>>>
```

```
>>> 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
```

With Python, it is possible to use the `**` operator to calculate powers [1]:

```
>>>
```

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

The equal sign (`=`) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>>
```

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>>
```

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result [2]. `\` can be used to escape quotes:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes.

```
>>>
```

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
>>> print('"Isn\'t," she said.')
"Isn't," she said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an `r` before the first quote:

```
>>>
```

```
>>> 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 'um', followed by 'iium'
>>> 3 * 'um' + 'iium'
'umumumiium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>>
```

```
>>> 'Py' 'thon'
'Python'
```

This only works with two literals though, not with variables or expressions:

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>>
```

```
>>> prefix = 'Py'

>>> prefix + 'thon'
'Python'
```

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 = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

Indices may also be negative numbers, to start counting from the right:

```
>>>
```

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
```

```
>>> word[-6]
'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, ***slicing*** allows you to obtain substring:

```
>>>
```

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

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
>>>
```

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>>
```

```
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
>>>
```

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
>>>
```

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
>>>
```

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
>>>
```

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

The built-in function `len()` returns the length of a string:

```
>>>
```

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Waiting for the User

The following line of the program displays the prompt and, the statement saying “Press the enter key to exit”, and then waits for the user to take action –

```
input("\n\nPress the enter key to exit.")
```

Here, "\n\n" is used to create two new lines before displaying the actual line. Once the user presses the key, the program ends. This is a nice trick to keep a console window open until the user is done with an application.

Your First Steps towards Programming

we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```
>>>
```

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
```

NEXT LAB

Python Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types –

- ✓ Numbers
- ✓ String
- List
- Tuple
- Dictionary