Bootcamp Training

Python

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Authorized & published by Summitworks Technologies Inc

Python – Day 1 Training Plan

- Overview of Python Features
- Getting and Installing Python
- Accessing Python Documentation: Python Enhancement
- Compiler and Interpreter
- IDEs and Notebooks
- Using Whitespace to Structure Programs
- Python Comments, Identifiers, Variables and Keywords
- Operators in Python
  - Unary and Binary Arithmetic Operations
  - Comparison and Boolean Operations
  - Identity and Membership Operators
  - Ternary Conditional Operator
  - Order of Operations and Operator Evaluation
- Selection Statement
  - if/elif/else Statements
- Repetition Statements
  - Creating Loops with while and for
  - else after Repetition Statements
  - Loop Modification with break and continue
  - for loop with range() function
  - Pass keyword in flow control statements
- Returning Values with return Statements
- Quotation Marks and Special Characters
- Manipulating Strings with String Methods
- Using the format() Function to Format Strings
- Python Functions
  - Defining Functions
  - Variables and scopes
  - Creating Anonymous Functions
  - Defining Functions with Arguments
  - Defining Flexible Functions that Take Variable Length Arguments
- Lambda Expressions
- Live Examples

Python Training Plan

Technology	Duration (Days)
Python	Day1 to Day 5 (5)
Django	Day 7 to 12 (6)
Web services and web scraping	Day 13 and14(2)
Mongo DB	Day 15(1)
Django Real time Project	Day 16 to 19 (4)
JavaScript Frameworks	Day 20 to Day 29(10)
JavaScript Project	Day 30 to 34(5)
AWS	Day 35 to 42 (8)

Training Objective

To Train fresh graduates or experienced professionals in IT Career Paths to facilitate their development towards becoming professionals in the IT Industry

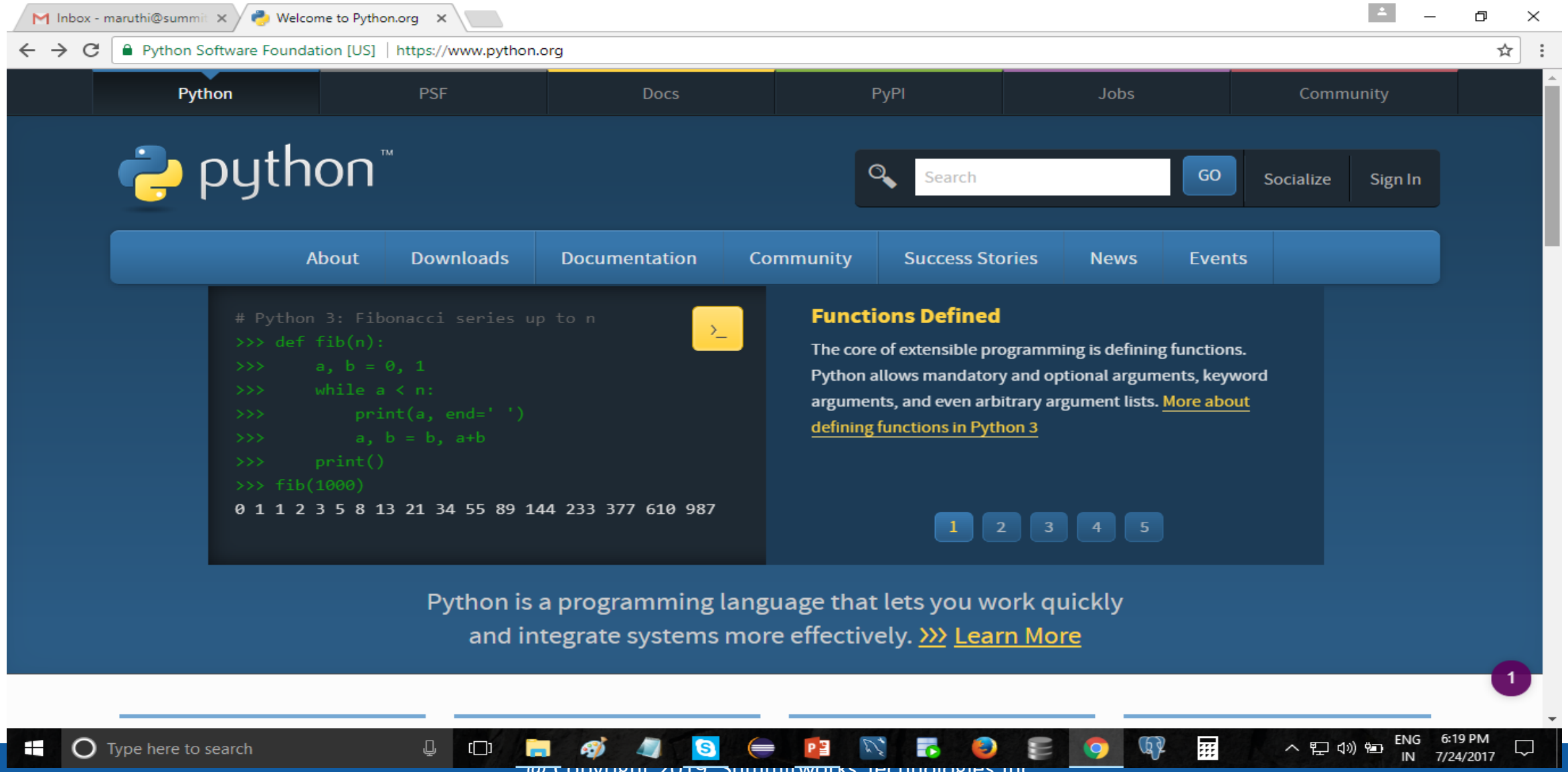
Training Highlights

- Industry standards based with real time concepts
- Harden skills through Hands-On and Project Work
- Flavored with the latest technology in software

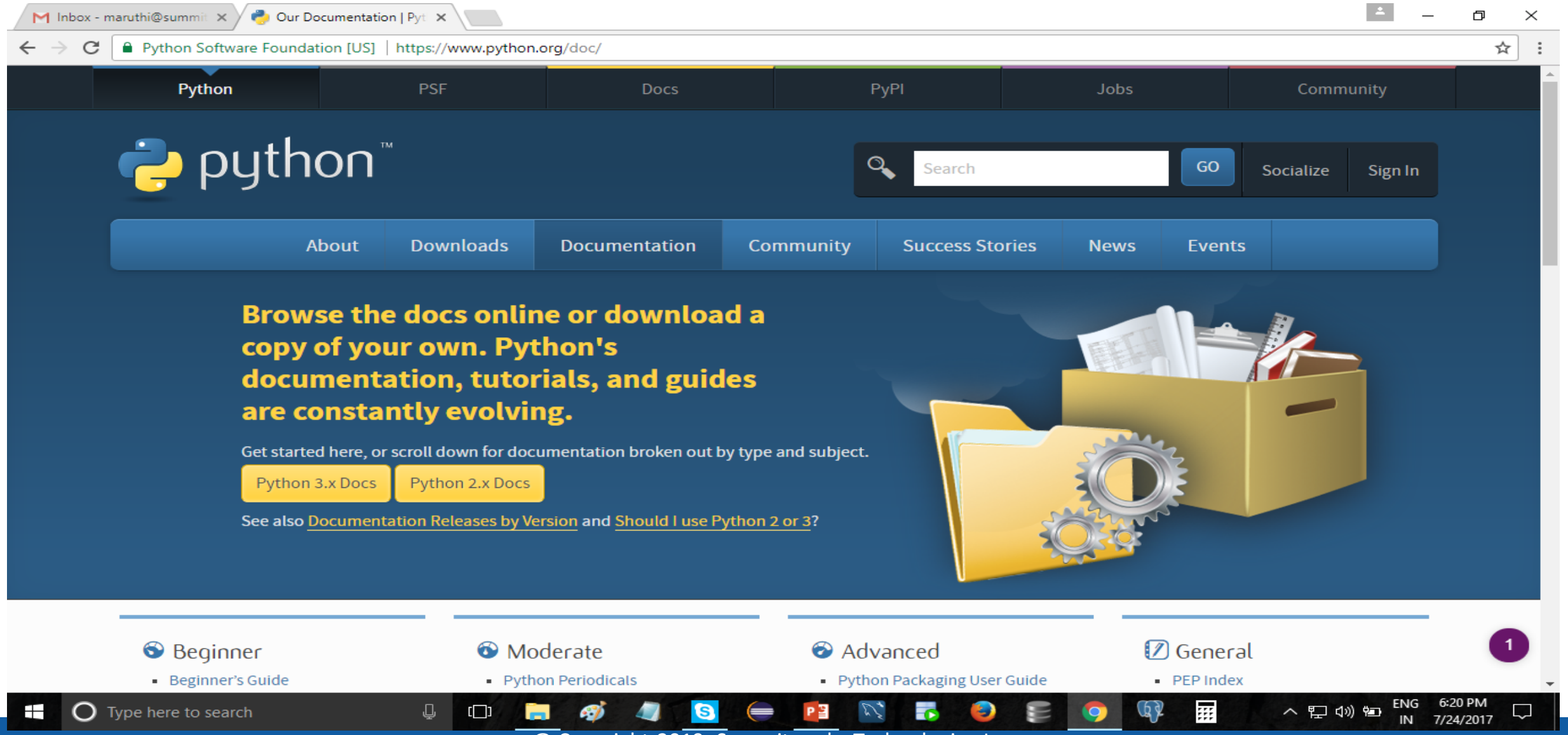
Brief History of Python

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python
- Open sourced from the beginning, managed by Python Software Foundation
- Considered a scripting language, but is much more Scalable, object oriented and functional from the beginning Used by Google from the beginning

http://python.org/



http://www.python.org/doc



Python features

Python features	
No-type declarations	cross-platform programming without ports
automatic memory management	built-in interfaces to external services
object-oriented programming	Support for web development using Django flask
rapid development cycle	interactive, dynamic nature
simpler, shorter, more flexible	Best for automation
mixed language systems	Support for Multithreading and Exception handling

What can you do with Python

Used to develop almost any kind of application using python

- Automation
  - Automate the deployment
  - Automate the cloud services(AWS,Azure)
  - Selenium testing
- Desktop applications by using tkinter , wxPython.
- Web development using Django / flask[server side programming]
- Applications of AI
- Data Science
- Reading web data automatically [web scraping]
- Web Services
- Scientific and mathematical programming [as python providing numpy and scipy packages]
- Rapid application development and many more…

Advantages of pip

- What are the advantages of using pip[python installer package]
- It’s a package management system from python
- We can install all required modules using pip command

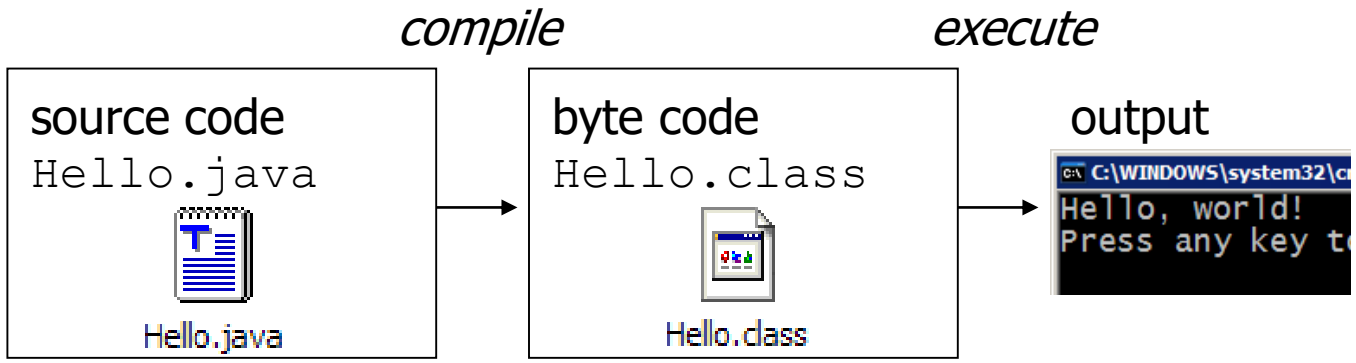
- Get help using below command
  - Pip help
- Install any package using below format
  - Pip install package-name
  - Ex: pip install django

Difference between Compiler and Interpreter

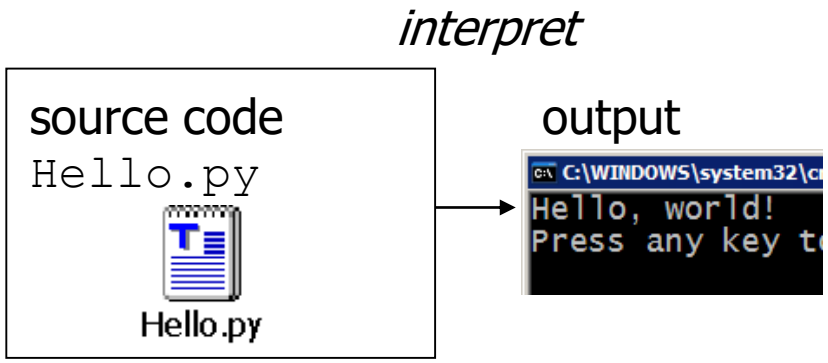
- Compiled languages are written in a code that can be **executed directly** on a computer's processor. A compiler is a special program that processes statements written in a particular programming language and turns them into **machine language** or "code" that a computer's processor uses.
- An **interpreted language** is any programming language that isn't already in "machine code" prior to runtime. Unlike **compiled languages** , an interpreted language's translation doesn't happen beforehand. Translation occurs at the same time as the program is being executed.

Compiling and Interpreting

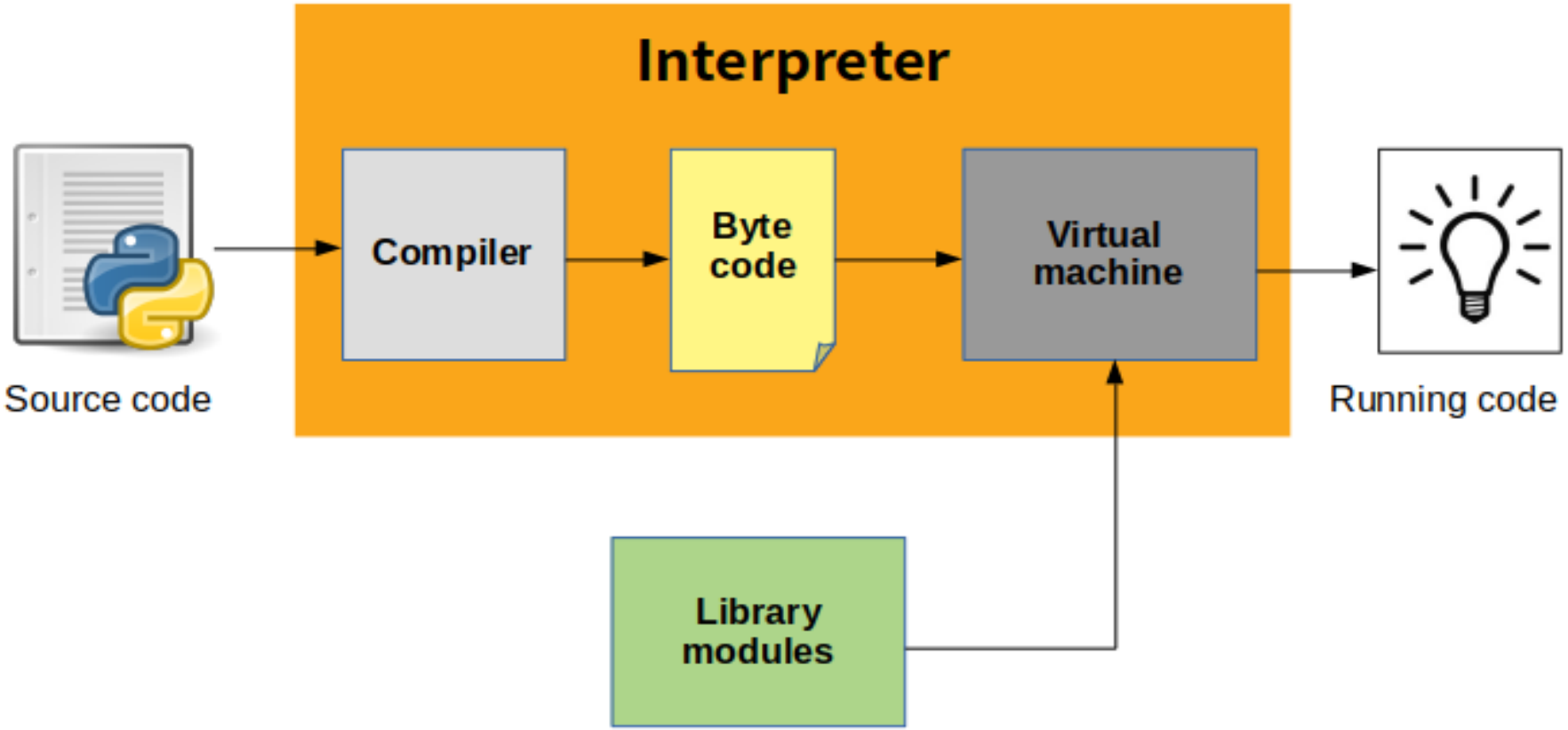
- Many languages require you to *compile* (translate) your program into a form that the machine understands.



- Python is instead directly *interpreted* into machine instructions.



Python Interpreter



Is Python Compiled or Interpreted?

- Python as a programming language has no saying about if it's an compiled or **interpreted** programming language. Python program runs directly from the **source code** . so, Python will fall under byte code interpreted. The .py source code is first compiled to byte code as .pyc. This byte code can be interpreted (official CPython), or JIT compiled. Python source code (.py) can be compiled to different byte code also like **IronPython** (.Net) or Jython (JVM). There are multiple implementations of **Python language** . The official one is a byte code interpreted one. There are byte code JIT compiled implementations too.
- So, Python**(Cpython)** is neither a true compiled time nor pure **interpreted language** but it is called interpreted language.

How Python is interpreted?

- The **python code** you write is compiled into python bytecode, which creates file with extension **.pyc** . The bytecode compilation happened internally, and almost completely hidden from developer. Compilation is simply a translation step, and byte code is a lower-level, and **platform-independent** , representation of your source code. Roughly, each of your source statements is translated into a group of byte code instructions. This byte code translation is performed to speed execution **byte code** can be run much quicker than the original source code statements.
- The **.pyc file** , created in compilation step, is then executed by appropriate virtual machines. The Virtual Machine just a big loop that iterates through your **byte code** instructions, one by one, to carry out their operations. The **Virtual Machine** is the runtime engine of Python and it is always present as part of the Python system, and is the component that truly runs the **Python scripts** . Technically, it's just the last step of what is called the Python interpreter.

Installing

- Python is pre-installed on most Unix systems, including Linux and MAC OS X
- The pre-installed version may not be the most recent
- Download latest version from http://python.org/download/
- Python comes with a large library of standard modules

Python IDEs and Shells

- There are many Integrated Development Environments
  - Pycharm [best one]
  - IDLE
  - Eclipse + PyDev
  - Spyder
- As well as enhanced shells
- The Jupyter Notebook

Advantage of using The Jupyter Notebook

- Its a interactive code environment for python.
- You can see the real-time result in the notebook.
- The Jupyter Notebook is a web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, machine learning and much more.

What is the difference between pycharm and jypyter note book

- Notebook:
  - Notebook has the ability to re-run individual snippets
  - In a notebook you can edit those snippets before re-running them.
  - We can deploy Jupyter Notebook on a remote server and access from web browser, so that your laptop won't get heavy running Python itself.
- Pycharm:
  - Pycharm supports multiple file types, with inspection and code completion in many of them, including python, json, rst and many others.
  - Pycharm has integration with config management
  - Pycharm has integration with debugging, code coverage and profilers.

What is a program?

- A program is a sequence of instructions
- The computer executes one after the other, as if they had been typed to the interpreter
- Saving as a program is better than re-typing from scratch

```
x = 1
y = 2
x + y
Print(x + y)
Print("The sum of", x, "and", y, "is", x+y)
```

First example

```
#!/usr/local/bin/python
# import systems module
Print(“hello, welcome to python programming”)
```

Identifiers and Keywords

- python keywords are the words that are reserved. That means you can't use them as name of any entities like variables, classes and functions.
- >>>help() # first type help() in python console and then type keywords to get python keywords
- Keywords

```
help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass
```

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).

- **Rules for writing Identifiers**

  - Identifiers can be combination of uppercase and lowercase letters, digits or an underscore(_). So **myVariable**, **variable_1**, **variable_for_print** all are valid python identifiers.
  - An Identifier can not start with digit. So while **variable1** is valid, **1variable** is not valid.
  - We can't use special symbols like !,#,@,%,$ etc in our Identifier.
  - Identifier can be of any length.

Indentation

- Indenting
  - Used to delimit code
  - Python uses no end of statement character
  - Therefore a new line of code is determined by return space
  - Indenting is the same way
    - Python does not use {} to enclose a multi-line statement
    - The indentation must be exactly the same same
  - There is no exact rule for the number of spaces but they are generally in groups of three

Expressions

- **expression**: A data value or set of operations to compute a value.

  Examples: `1 + 4 * 3`

  `42`

- Arithmetic operators we will use:

`+ - * /`	addition, subtraction/negation, multiplication, division
`%`	modulus, a.k.a. remainder
`**`	exponentiation

- **precedence**: Order in which operations are computed.
  - `* / % **` have a higher precedence than `+ -`

    `1 + 3 * 4` is `13`

  - Parentheses can be used to force a certain order of evaluation.

    `(1 + 3) * 4` is `16`

Type of operators in Python

- Arithmetic operators
- Comparison (Relational) operators
- Logical (Boolean) operators
- Bitwise operators
- Assignment operators
- Special operators

Arithmetic

- Symbols
  - * = multiply
  - / = divide
  - % = modulus
  - ** = exponential
  - // = floor division
- Order
  - Operators are done in order of parenthesis, exponents, multiple and divide (left to right), and lastly add and subtract (left to right)

Arithmetic

Python operation	Arithmetic operator	Algebraic expression	Python expression
Addition	+	$f + 7$	`f + 7`
Subtraction	-	$p - c$	`p - c`
Multiplication	*	bm	`b * m`
Exponentiation	**	x^y	`x ** y`
Division	/ // (new in Python 2.2)	x / y or <Anchor4> or $x \quad y$	`x / y` `x // y`
Modulus	%	$r \bmod s$	`r % s`

Order of Evaluation

Operator(s)	Operation(s)	Order of Evaluation (Precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
**	Exponentiation	Evaluated second. If there are several, they are evaluated right to left.
* / // %	Multiplication Division Modulus	Evaluated third. If there are several, they are evaluated left to right. [*Note:* The // operator is new in version 2.2]
+ -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Order of Evaluation: Example

Step 1.

```
y = 2 * 5 ** 2 + 3 * 5 + 7
      5 ** 2 = 25
```

Exponentiation

Step 2.

```
y = 2 * 25 + 3 * 5 + 7
    2 * 25 = 50
```

Leftmost multiplication

Step 3.

```
y = 50 + 3 * 5 + 7
        3 * 5 = 15
```

Multiplication before addition

Step 4.

```
y = 50 + 15 + 7
    50 + 15 = 65
```

Leftmost addition

Step 5.

```
y = 65 + 7 is 72
```

Last edition

Step 6.

```
y = 72
```

Python assigns **72** to **y**

Order in which a second-degree polynomial is evaluated

Logical operators

operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Bitwise operators

Bitwise operators act on operands as if they were string of binary digits. It operates bit by bit, hence the name.
For example, 2 is 10 in binary and 7 is 111.
In the table below: Let *x* = 10 (0000 1010 in binary) and *y* = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x& y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x>> 2 = 2 (0000 0010)
<<	Bitwise left shift	x<< 2 = 40 (0010 1000)

Identity operators

Python language offers some special type of operators like the identity operator or the membership operator.

is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

```
>>> p = 'hello'
>>> ps = p
>>> ps is p
True
```

Identity operators

```
In [29]: x = [1,2,3]
         y = [1,2,3]
         x is y

Out[29]: False

In [30]: id(x)

Out[30]: 2300560976328

In [31]: id(y)

Out[31]: 2300561749448

In [32]: x==y

Out[32]: True
```

Identity operators

```
a = 0
a+=1 # it will adjust the mory for ,since they are small [-5 to 256 ]
b=1 # auto intern
print(a)
print(b)
a is b
```

```
1
1

True
```

```
a = 1000
a+=1
b = 1001 # these are big number after 256
print(a)
print(b)
a is b
```

```
1001
1001

False
```

```
a==b
```

```
True
```

Identity operators

```
x = 'a'
x+='bc'
y = 'abc'
x is y
```

```
False
```

```
x==y # value comparision
```

```
True
```

Membership operators

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).
In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

```
>>> 'good' in 'this is a great example'
False
>>> 'good' not in 'this is a great example'
True
```

Chaining comparison operators

Comparison operators can be chained. Consider the following examples:

```
>>> x = 2
>>> 1 < x < 3
True
>>> 10 < x < 20
False
>>> 3 > x <= 2
True
>>> 2 == x < 4
True
```

Operators in Python

Operators	Description
lambda	Lambda Expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT

Operators	Description
in, not in	Membership tests
is, is not	Identity tests
<, <=, >, >=, !=, ==	Comparisons

Operators	Description
x.attribute	Attribute reference
x[index]	Subscription
x[index:index]	Slicing
f(arguments, ...)	Function call

Operators	Description
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and Subtraction

Operators	Description
*, /, %	Multiplication, Division and Remainder
+x, -x	Positive, Negative
~x	Bitwise NOT
**	Exponentiation

Operators	Description
(expressions, ...)	Binding or tuple display
[expressions, ...]	List display
{key:datum, ...}	Dictionary display
`expressions, ...`	String conversion

+= but not ++

- Python has incorporated operators like +=, but ++ (or --) do not work in Python

difference between `==` and `is` in Python?

- == is for value equality. Use it when you would like to know if two objects have the same value.
- **is** is for reference equality. Use it when you would like to know if two references refer to the same object.

Ternary operator

- A ternary operator is a simple terse conditional assignment statement.
- exp1 if condition else exp2
- If condition is true, exp1 is evaluated and the result is returned. If the condition is false, exp2 is evaluated and its result is returned.

Comments

- Anything after a # symbol is treated as a comment
- This is just like Perl

Types of values

- Integers (**`int`**): **`-22`**, **`0`**, **`44`**
  - Arithmetic is exact
  - Some funny representations: **`12345678901L`**
- Real numbers (**`float`**, for “floating point”): **`2.718`**, **`3.1415`**
  - Arithmetic is approximate, e.g., **`6.022*10**23`**
  - Some funny representations: **`6.022e+23`**
- Strings (**`str`**): **`"I love Python"`**, **`""`**
- Truth values (**`bool`**, for “Boolean”): **`True`**, **`False`**

Lambda Functions

- Python supports an interesting syntax that lets you define one-line mini-functions on the fly. Borrowed from Lisp, these so-called lambda functions can be used anywhere a function is required.

Introducing lambda Functions

- >>> def f(x):
- ... return x*2
- ...
- >>> f(3)
- 6
- >>> g = lambda x: x*2 ------ 1
- >>> g(3)
- 6
- >>> (lambda x: x*2)(3) 2 ------ 2
- 6

Lambda Functions

- This is a lambda function that accomplishes the same thing as the normal function above it. Note the abbreviated syntax here: there are no parentheses around the argument list, and the return keyword is missing (it is implied, since the entire function can only be one expression). Also, the function has no name, but it can be called through the variable it is assigned to.
- You can use a lambda function without even assigning it to a variable. This may not be the most useful thing in the world, but it just goes to show that a lambda is just an in-line function.

Control Structure

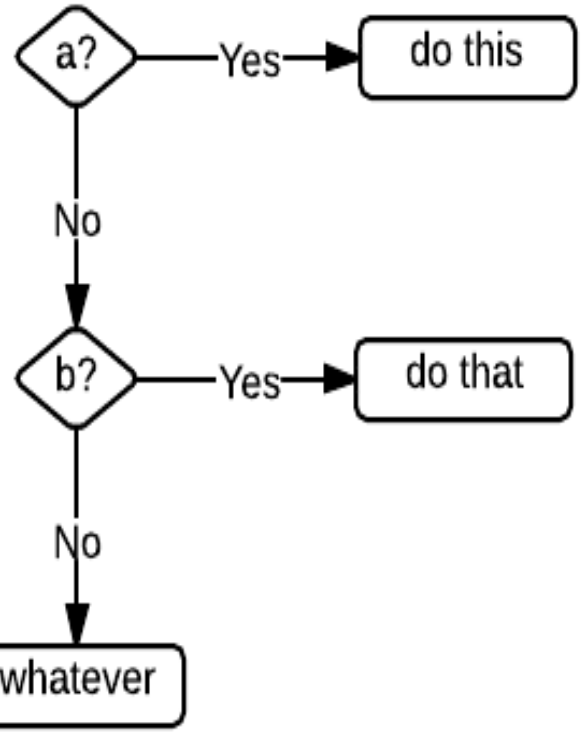
- Sequential order
  - Statements are executed in the order they are written
- Selection structure
  - The **`if`** statement
  - The **`if/else`** statement
  - The **`if/elif/else`** statement
- Repetition structure
  - The **`while`** repetition structure
  - The **`for`** repetition structure

`if/else` and `if/elif/else` Selection Structures

- The if statement
- The **`if/else`** statement
  - Double selection statement
  - Allows the programmer to perform an action when a condition is true
  - An alternate action is preformed when the action is false
- The **`if/elif/else`** statement
  - Multiple selection statement
  - This is used in place of nested **`if/else`** statements
  - The final **`else`** statement is optional
    - It is used as a default action should all other statements be false

syntax

```
if a:
    do this
elif b:
    do that
else:
    whatever
```



- Core Python does not provide switch or case statements as in other languages, but we can use

if..elif...statements to simulate switch case

Nested if statements

```
If expression1:
        statement(s)
        if expression2:
                statement(s)
        elif expression3:
                statement(s)
        else
                statement(s)
elif expression4:
        statement(s)
else:
        statement(s)
```

Note: left side spaces are very important

`while` Repetition Structure

- Repetition Structures
  - Allow a program to repeat an action while a statement is true
- Using **`while`** Repetition
  - The action is contained within the body of the loop
    - Can be one or more than one action
  - Condition should evaluate to false at some point
    - Creates a infinite loop and program hangs

The else Statement Used with Loops

- Python supports to have an **else** statement associated with a loop statement.
- • If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- • If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Avoid else Blocks After for and while Loops

- Python loops have an extra feature that is not available in most other programming languages: you can put an else block immediately after a loop's repeated interior block.
- Using a break statement in a loop will actually skip the else block.
- The else block after a loop only runs if the loop body did not encounter a break statement.

`for` Repetition Structure

- The for loop
  - Function range is used to create a list of values
    - **`range`** (integer)
      - Values go from 0 u to given integer
    - **`range`** (integer, integer)
      - Values go from first up to second integer
    - **`range`** (integer, integer, integer)
      - Values go from first up to second integer but increases in intervals of the third integer
  - The loop will execute as many times as the value passed
  - **`for` *`counter`* `in range (` *`value`* `)`**

Examples

- print(list(range(10)))

- # Output: [2, 3, 4, 5, 6, 7]
- print(list(range(2, 8)))

- # Output: [2, 5, 8, 11, 14, 17]
- print(list(range(2, 20, 3)))

Output: ?

The else Statement Used with for Loops

- Python supports to have an **else** statement associated with a loop statement.
- • If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
- • If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

`break` and `continue` Statements

- The **`break`** statement
  - Used to make a loop stop looping
  - The loop is exited and no more loop code is executed
- The **`continue`** statement
  - Used to continue the looping process
  - All following actions in the loop are not executed
    - But the loop will continue to run

Example

```
1  # Fig. 3.24: fig03_24.py
2  # Using the break statement in a for structure.
3
4  for x in range( 1, 11 ):
5
6     if x == 5:
7        break
8
9     print x,
10
11 print "\nBroke out of loop at x =", x
```

The loop will go from 1 to 10

When x equals 5 the loop breaks. Only up to 4 will be displayed

Shows that the counter does not get to 10 like it normally would have

The pass Statement

- The pass statement does nothing.
- It can be used when a statement is required syntactically but the program requires no action.
- For example:

```
while True:
    pass        # Busy-wait for keyboard interrupt
```

break and else in python

No ,it wont execute else block

Python Strings

- Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.
- Creating strings is as simple as assigning a value to a variable. For example:
- var1 = 'Hello World!'
- var2 = "Python Programming"

Accessing Values in Strings:

- Python does not support a character type; these are treated as strings of length one, thus also considered a substring.
- To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

```
var1 = 'Hello World'

print ("var1[0]: ", var1[0])
```

```
var1[0]:  H
```

String Special Operators:

- Assume string variable a holds 'Hello' and variable b holds 'Python', then:
- **+ Concatenation** - Adds values on either side of the operator a + b will give Hello Python
- *** Repetition** - Creates new strings, concatenating multiple copies of the same string a*2 will give -HelloHello
- **[] Slice** - Gives the character from the given index a[1] will give **e**

String Special Operators:

- **[:] Range Slice** - Gives the characters from the given range a[1:4] will give **ell**
- **In Membership** - Returns true if a character exists in the given string **H in a** will give 1
- **not in Membership** - Returns true if a character does not exist in the given string **M not in a** will give 1

String Formatting Operator:

- One of Python's coolest features is the string format operator **%**. This operator is unique to strings and makes up for the pack of having functions from C's printf() family
- Following is a simple example:

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % (‘Sara', 42)
```

- When the above code is executed, it produces the following result:

```
My name is Sara and weight is 42 kg!
```

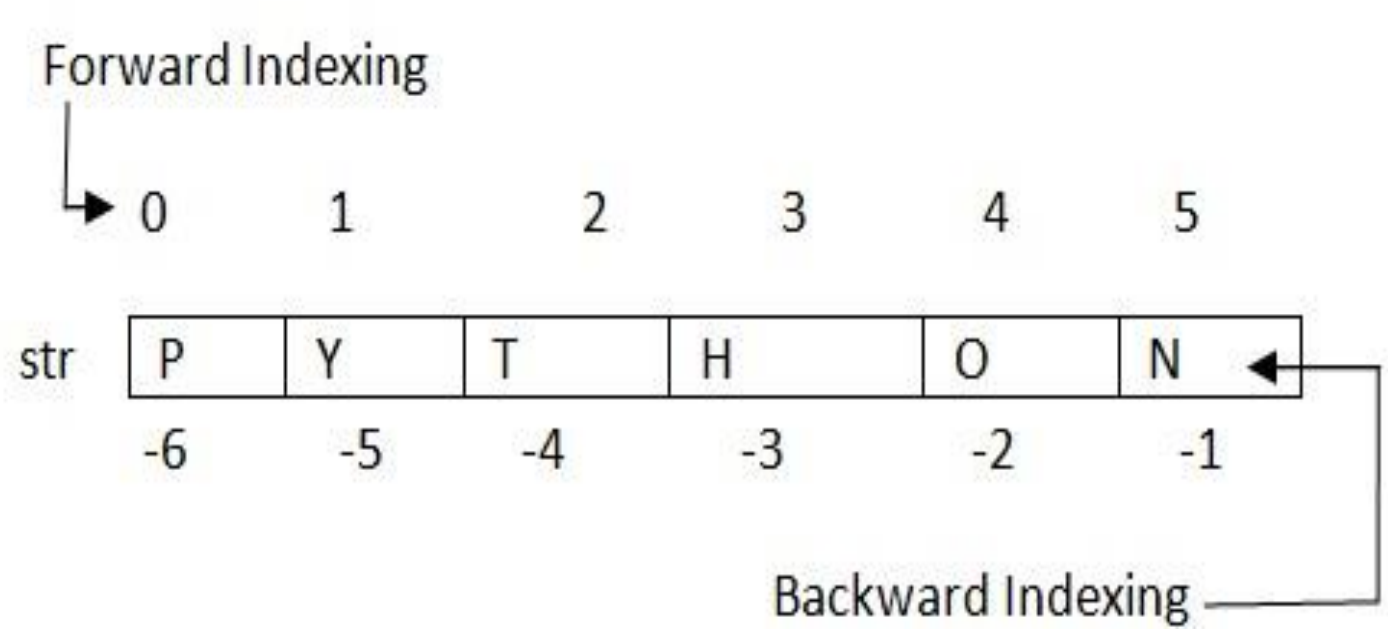
Examples

- Here is the list of complete set of symbols, which can be used along with %:
  - %c character
  - %s string conversion via str() prior to formatting
  - %i signed decimal integer
  - %d signed decimal integer
- %u unsigned decimal integer
- %o octal integer
- %x hexadecimal integer (lowercase letters)
- %X hexadecimal integer (UPPERcase letters)
- %e exponential notation (with lowercase 'e')
- %E exponential notation (with UPPERcase 'E')
- %f floating point real number
- %g the shorter of %f and %e
- %G the shorter of %f and %E

Accessing Strings:

- In Python, Strings are stored as individual characters in a contiguous memory location.
- The benefit of using String is that it can be accessed from both the directions in forward and backward.
- Both forward as well as backward indexing are provided using Strings in Python.
  - Forward indexing starts with 0,1,2,3,....
  - Backward indexing starts with -1,-2,-3,-4,....

Strings index



Sample code

```
name="Summit"
length=len(name)
i=0
for n in range(-1,(-length-1),-1):
    print name[i],"\t",name[n]
    i+=1
```

How to change or delete a string?

- Strings are immutable. This means that elements of a string cannot be changed once it has been assigned. We can simply reassign different strings to the same name.
- What is the output for the given code

'

```
my_string = 'program'
my_string[5] = 'a’
```

TypeError: 'str' object does not support item assignment

But how to change elements of string as its imutable in python

- **Method 1**
- Given by this answer
- text = 'abcdefg'
- new = list(text)
- new[6] = 'W'
- ''.join(new)
- Which is pretty slow compared to 'Method 2'
- timeit.timeit("text = 'abcdefg'; s = list(text); s[6] = 'W'; ''.join(s)", number=1000000)
- 1.041158199310302 7
- **Method 2 (FAST METHOD) [using +]**
- text = 'abcdefg'
- text = text[:1] + 'Z' + text[2:]
- Which is much faster:
- timeit.timeit("text = 'abcdefg'; text = text[:1] + 'Z' + text[2:]", number=1000000)
- 0.3465199470520019 5

But how to change elements of string as its imutable in python

Method 3:

- Byte array:
- timeit.timeit("text = 'abcdefg'; s = bytearray(text); s[1] = 'Z'; str(s)", number=1000000)
- 1.0387420654296875

Iterating Through String

- Using for loop we can iterate through a string. Here is an example to count the number of 'l' in a string.

```
count = 0
for letter in 'Hello World':
        if(letter == 'l'):
                count += 1
print(count,'letters found')
```

String Membership Test

- >>> 'a' in 'program'
- True
- >>> 'at' not in 'battle'
- False

Built-in functions to Work with Python

- Various built-in functions that work with sequence, works with string as well.

- Some of the commonly used ones are enumerate() and len(). The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Python String Formatting

- If we want to print a text like –

He said, "What's there?"-

can we neither use single quote or double quotes? Will it work?

Solution

- One way to get around this problem is to use triple quotes. Alternatively, we can use escape sequences.

- An escape sequence starts with a backslash and is interpreted differently. If we use single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes. Here is how it can be done to represent the above text.

Common Python String Methods

- There are numerous methods available with the string object. The format() method that we mentioned above is one of them. Some of the commonly used methods are lower(), upper(), join(), split(), find(), replace() etc.
- Check the document for more methods

Common Python String Methods

```
s = 'hello python'
```

```
s="hello python"
```

```
s.capitalize()
```

```
'Hello python'
```

```
s.upper()
```

```
'HELLO PYTHON'
```

```
s.lower()
```

```
'hello python'
```

```
s.count('o')
```

```
2
```

```
s.find('o')
```

```
4
```

```
s.center(20,' ')
```

```
'    hello python    '
```

```
s.isalnum()
```

```
False
```

```
s
```

```
'hello python'
```

```
s[5]
```

```
' '
```

```
s[-1]
```

```
'n'
```

```
s*2
```

```
'hello pythonhello python'
```

```
s+'gello'
```

```
'hello pythongello'
```

Common Python String Methods

```
s[2]
```

```
'l'
```

```
s[2:7]
```

```
'llo p'
```

```
s[2:] #range slice
```

```
'llo python'
```

```
s[:2]
```

```
'he'
```

```
s
```

```
'hello python'
```

```
s[::-1] # begin,end,step
```

```
'nohtyp olleh'
```

```
s
```

```
'hello python'
```

S=‘hello python’

Python Functions

- Functions are reusable components of programs.
- They allow us to give a name to a block of statements.
- We can execute that block of statements by just using that name anywhere in our program and any number of times.
- This is known as *calling/invoking* the function.

Python Functions

- Functions are defined using the **def** keyword.
- This is followed by an *identifier* name for the function name.
- This is followed by a pair of parentheses which may enclose some names of variables.
- The line ends with a colon and this is followed by a new block of statements which forms the body of the function with proper indentation.

Defining a function

Syntax

```
def sayHello():
      print 'Hello World!'  # A new block
      # End of the function
sayHello() # call the function
```

Defining a Function

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or ***docstring***.
- The code block within every function starts with a colon **(:)** and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Function Parameters

- values we supply to the function to perform any task.
- Specified within the pair of parentheses in the function definition, separated by commas.
- When we call the function, we supply the values in the same way and order.
- the names given in the function definition are called *parameters.*
- the values we supply in the function call are called *arguments*.
- Arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object).

Using Function Parameters

```
# Demonstrating Function Parameters
def printMax(a, b):
if a > b:
print a, 'is maximum'
else:
print b, 'is maximum'
printMax(3, 4) # Directly give literal values
x = -5
y = -7
printMax(x, y) # Give variables as arguments
```

Local and Global Variables

- Variable names initialized and used inside one function are invisible to other functions. Such variables are called local variables
- While declaring variables inside a function definition:
  - They are not related in any way to other variables with the same names used outside the function
  - That is, variable declarations are local to the function.
- This is called the *scope* of the variable.
- All variables have the scope of the block they are declared in, starting from the point of definition of the variable.

Using Local Variables

```
# Demonstrating local variables
def func(x):
print 'Local x is', x
x = 2
print 'Changed local x to', x
x = 50
func(x)
print 'x is still', x
```

Example

```
def main():
    x = 3
    f()

def f():
    print(x)  # error: f does not know about the x defined in
main

main()
```

solution

```
def main():
    x = 3
    f(x)

def f(x):
    print(x)

main()
```

Global Variables

- If you define *global variables* (variables defined outside of any function definition), they are visible inside all of your functions. They have *global scope*.
- It is good programming practice to avoid defining global variables and instead to put your variables inside functions and explicitly pass them as parameters where needed. One common exception is constants.
- A *constant* is a name that you give a fixed data value to, by assigning a value to the name only in a single assignment statement. You can then use the name of the fixed data value in expressions later.

Using global variables

```
# demonstrating global variables
def func():
        global x
        print 'x is', x
        x = 2
        print 'Changed x to', x
x = 50
func()
print 'Value of x is', x
```

The return Statement

- The return statement is used to return from a function
  i.e. break out of the function.
- We can optionally return a value from the function as well.
- Note that a return statement without a value is equivalent to return None.
- None is a special value in Python which presents nothingness.
- For example, it is used to indicate that a variable has no value if the variable has a value of None.
- Every function implicitly contains a return None statement.
- We can see this by running print someFunction() where the function *someFunction* does not use the return statement such as
- def someFunction():
  - pass

The return Statement

```
# Demonstrating the return Statement
def max(x, y):
                    if x > y:
                                return x
                    else:
                                return y
print max(2, 3)
```

The *Anonymous* Functions

- These functions are called anonymous because they are not declared in the standard manner by using the ***def*** keyword. You can use the *lambda* keyword to create small anonymous functions.
- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

- The syntax of *lambda* functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]:expression

Function Arguments

You can call a function by using the following types of formal arguments:

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

- Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.
- Example:

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;

# Now you can call printme function
printme()   # it’s a error, as you did not passed the string in the specific order
Printme(“hello”) # it print the string you are passing
```

Keyword arguments

- Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.
- This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters.

```
# Function definition is here
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;
# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

Default arguments

- A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments.

```
# Function definition is here
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

Variable-length arguments

- You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments.

Syntax

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all non keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call.

Any Queries