BCSE101E

Computer Programming: Python

Course Faculty: Dr. Rajesh M – SCOPE, VIT Chennai .

Module-2 Python Programming Fundamentals

Need of programming Languages

- Computers can execute tasks very rapidly and assist humans
- Programming languages – Helps for communication between human and machines.
- They can handle a greater amount of input data.
- But they cannot design a strategy to solve problems for you

Python – Why?

- Simple syntax
- Programs are clear and easy to read
- Has powerful programming features
- Companies and organizations that use Python include YouTube, Google, Yahoo, and NASA.
- Python is well supported and freely available at www.python.org.

Python – Why?

- **Guido van Rossum** – Creator
- Released in the early **1990s.**
- Its name comes from a **1970s British comedy sketch television show** called ***Monty Python's Flying Circus***.

Python….

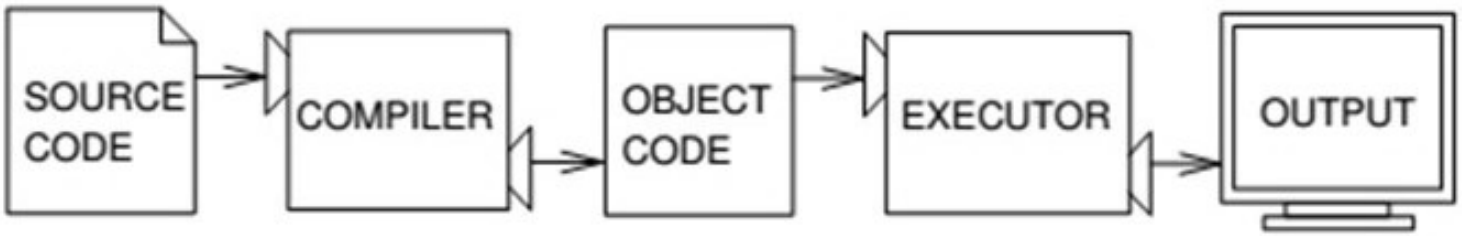
- **High-level language**; other high-level languages you might have heard of are C, C++, Perl, and Java.
- There are also **low-level languages**, sometimes referred to as “machine languages” or “assembly languages.”
- Computers can only run programs written in low-level languages.
- So programs written in a high-level language have to be processed before they can run.

Python….

- Two kinds of program translator to convert from high-level languages into low-level languages:
  - **Interpreters**
  - **Compilers**.
- An interpreter processes the program by reading it line by line

Python….

- Compiler translates completely a high level program to low level it completely before the program starts running.
- High-level program is called **source code**
- Translated program is called the **object code** or the **executable**.



A compiler translates source code into object code, which is run by a hardware executor.

Python….

- Python is considered an interpreted language because Python programs are executed by an interpreter.
- There are two ways to use the interpreter:

  interactive mode and **script mode**.

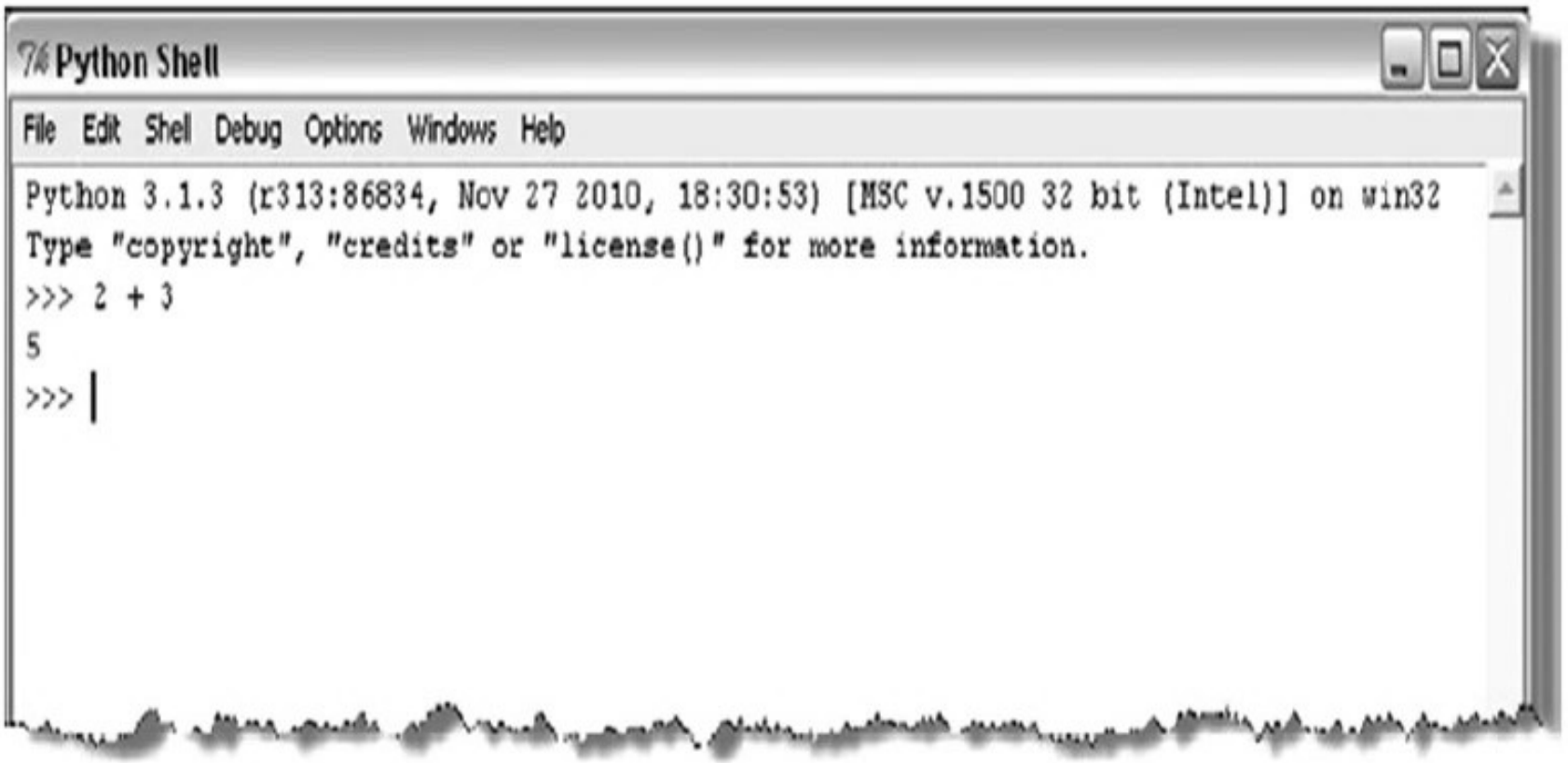
Python....Interactive mode

In interactive mode, you type Python programs and the interpreter displays the result:

```
>>> 1 + 1
2
```

The shell prompt, >>>, is the **prompt** the interpreter uses to indicate that it is ready. If you type 1 + 1, the interpreter replies 2.

Python …. Shell



Python…. Script Mode

- Can also store code in a file and use the interpreter to execute the contents of the file, which is called a **script**.
- Python scripts have names that end with .py.
- Interactive mode is convenient for testing small pieces of code because you can type and execute them immediately.
- But for anything more than a few lines, should save your code as a script so you can modify and execute it in future.

Python....Script Mode ...

File name : first.py

```
print(4+3)
print(4-3)
print(4>3)
print("hello World")
```

```
C:\Python34\python.exe C:/Users/sathisbsk/first.py
7
1
True
hello World

Process finished with exit code 0
```

Problem

- Little Bob loves chocolate, and he goes to a store with Rs. *N* in his pocket. The price of each chocolate is Rs. *C*. The store offers a discount: for every *M* wrappers he gives to the store, he gets one chocolate for free. This offer is available only once. How many chocolates does Bob get to eat?

PAC For Chocolate Problem

Input	Processing	Output
Amount in hand, N Price of one chocolate, C Number of wrappers for a free chocolate, M	Number of Chocolates P = Quotient of N / C Free chocolate F = Quotient of P/M	Total number of chocolates got by Bob

Pseudocode

- READ N and C
- COMPUTE num_of_chocolates as N/C
- CALCULATE returning_wrapper as number of chocolates/m
- TRUNCATE decimal part of returning_wrapper
- COMPUTE Chocolates_recieved as num_of_chocolates + returning_wrapper
- PRINT Chocolates_recieved

Knowledge Required

- Following knowledge is required in Python to write a code to solve the above problem
- Read input from user
- Data types in Python
- Perform arithmetic calculations
- Write output

What is an Identifier?

- An **identifier** is a sequence of one or more characters used to name a given program element.
- In Python, an identifier may contain letters and digits, but cannot begin with a digit.
- Special underscore character can also be used
- Example : line, salary, emp1, emp_salary

Rules for Identifier

- Python is ***case sensitive***, thus, Line is different from line.
- Identifiers may contain letters and digits, but cannot begin with a digit.
- The underscore character, _, is also allowed to aid in the readability of long identifier names. It should not be used as the *first* character
- Spaces are not allowed as part of an identifier.

Identifier Naming

Valid Identifiers		Invalid Identifiers	Reason Invalid
totalSales		'totalSales'	quotes not allowed
totalsales		total sales	spaces not allowed
salesFor2010		2010Sales	cannot begin with a digit
sales_for_2010		_2010Sales	should not begin with an underscore

Keywords

- A **keyword** is an identifier that has pre-defined meaning in a programming language.
- Therefore, keywords cannot be used as "regular" identifiers. Doing so will result in a syntax error, as demonstrated in the attempted assignment to keyword and below,

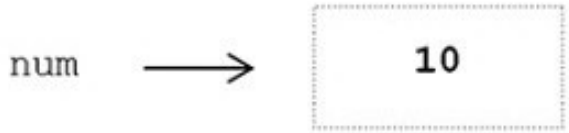
```
>>> and = 10
SyntaxError: invalid syntax
```

Keywords in Python

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

Variables and Identifiers

- A **variable** is a name (identifier) that is associated with a value.
- A simple description of a variable is "a name that is assigned to a value,"



Variables are assigned values by use of the **assignment operator**,

num = 10

num = num + 1

Comments

- Meant as documentation for anyone reading the code
- Single-line comments begin with the hash character ("#") and are terminated by the end of line.
- Python ignores all text that comes after # to end of line
- Comments spanning more than one line are achieved by inserting a multi-line string (with """ as the delimiter one each end) that is not used in assignment or otherwise evaluated, but sits in between other statements.
- *#This is also a comment in Python*
- """ This is an example of a multiline comment that spans multiple lines ... "“”

Literals

- A **literal** is a sequence of one or more characters that stands for itself.

Numeric literal

- A **numeric literal** is a literal containing only the digits 0–9, an optional sign character (1 or 2), and a possible decimal point. (The letter *e* is also used in exponential notation).
- If a numeric literal contains a decimal point, then it denotes a **floating-point value**, or "**float**" (e.g., 10.24); otherwise, it denotes an **integer value** (e.g., 10).
- *Commas are never used in numeric literals*

Numeric Literals in Python

Numeric Literals		
integer values	floating-point values	incorrect
5	5. 5.0 5.125 0.0005 5000.125	5,000.125
2500	2500. 2500.0 2500.125	2,500 2,500.125
+2500	+2500. +2500.0 +2500.125	+2,500 +2,500.125
-2500	-2500. -2500.0 -2500.125	-2,500 -2,500.125

Since numeric literals without a provided sign character denote positive values, an explicit positive sign character is rarely used.

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> 1024
???

>>> -1024
???

>>> .1024
???

>>> 1,024
???

>>> 0.1024
???

>>> 1,024.46
???
```

String Literals

- **String literals**, or "**strings**," represent a sequence of characters, 'Hello' 'Smith, John' "Baltimore, Maryland 21210"
- In Python, string literals may be surrounded by a matching pair of either single (') or double (") quotes.
- >>> print('Welcome to Python!')
  >>>Welcome to Python!

String Literal Values

`'A'`	- a string consisting of a single character
`'jsmith16@mycollege.edu'`	- a string containing non-letter characters
`"Jennifer Smith's Friend"`	- a string containing a single quote character
`' '`	- a string containing a single blank character
`''`	- the empty string

Note

If this string were delimited with single quotes, the apostrophe (single quote) would be considered the matching closing quote of the opening quote, leaving the last final quote unmatched,

`'Jennifer Smith's Friend'` ... *matching quote?*

Thus, Python allows the use of more than one type of quote for such situations. (The convention used in the text will be to use single quotes for delimiting strings, and only use double quotes when needed.)

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> print('Hello')
???

>>> print("Hello")
???

>>> print('Let's Go')
???

>>> print("Hello")
???

>>> print("Let's Go!')
???

>>> print("Let's go!")
???
```

Control Characters

- **S**pecial characters that are not displayed on the screen. Rather, they *control* the display of output
- Do not have a corresponding keyboard character
- Therefore, they are represented by a combination of characters called an *escape sequence*.
- The backslash (\) serves as the escape character in Python.
- For example, the escape sequence '\n', represents the *newline control character*, used to begin a new screen line

```
print('Hello\nJennifer Smith')
```

which is displayed as follows,

```
Hello
Jennifer Smith
```

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> print('Hello World')
???

>>> print('Hello World\n')
???

>>> print('Hello World\n\n')
???

>>> print('\nHello World')
???
```

```
>>> print('Hello\nWorld')
???

>>> print('Hello\n\nWorld')
???

>>> print(1, '\n', 2, '\n', 3)
???

>>> print('\n', 1, '\n', 2, '\n', 3)
???
```

Data Types

- Python's data types are built in the core of the language
- They are easy to use and straightforward.
- Data types supported by Python
  - **Boolean values**
  - **None**
  - **Numbers**
  - **Strings**
  - **Tuples**
  - **Lists**
  - **Sets**

Boolean values

- Primitive datatype having one of two values: True or False
- some common values that are considered to be True or False

Boolean values

print bool(True)	True
print bool(False)	False
print bool("text")	True
print bool("")	False
print bool(' ')	True
print bool(0)	False
print bool()	False
print bool(3)	True
print bool(None)	False

None

- Special data type - None
- Basically, the data type means non existent, not known or empty
- Can be used to check for emptiness

Numbers

Types of numbers supported by Python:

- Integers
- floating point numbers
- complex numbers
- Fractional numbers

Integers

- Integers have no fractional part in the number
- Integer type automatically provides extra precision for large numbers like this when needed (different in Python 2.X)
- >>> a = 10
- >>> b = a

Binary, Octal and Hex Literals

- 0b1, 0b10000, 0b11111111 # Binary literals: base 2, digits 0-1
- 0o1, 0o20, 0o377
- # Octal literals: base 8, digits 0-7
- 0x01, 0x10, 0xFF # Hex literals: base 16, digits 0-9/A-F
- (1, 16, 255)

Conversion between different bases

- Provides built-in functions that allow you to convert integers to other bases' digit strings
- oct(64), hex(64), bin(64)
- # Numbers=>digit strings ('0o100', '0x40', '0b1000000')
- These literals can produce arbitrarily long integers

Numbers can be very long

- >>> X = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFF
- >>> X
  5192296858534827628530496329220095
- >>> oct(X)
  '0o1777777777777777777777777777777777‘
- >>> bin(X)
  '0b11111111111111111111111111111111111111111111111111111111‘
- ...and so on... 11111‘

Floating Point Numbers

- Number with fractional part
- >>> 3.1415 * 2
- >>>6.283

Floating Point Numbers

Table 5-1. Numeric literals and constructors

Literal	Interpretation
`1234, -24, 0, 99999999999999`	Integers (unlimited size)
`1.23, 1., 3.14e-10, 4E210, 4.0e+210`	Floating-point numbers
`0o177, 0x9ff, 0b101010`	Octal, hex, and binary literals in 3.X
`0177, 0o177, 0x9ff, 0b101010`	Octal, octal, hex, and binary literals in 2.X
`3+4j, 3.0+4.0j, 3J`	Complex number literals
`set('spam'), {1, 2, 3, 4}`	Sets: 2.X and 3.X construction forms
`Decimal('1.0'), Fraction(1, 3)`	Decimal and fraction extension types
`bool(X), True, False`	Boolean type and constants

Arithmetic overflow

- a condition that occurs when a calculated result is too large in magnitude (size) to be represented,
  >>>1.5e200 * 2.0e210
  >>> inf
  This results in the special value inf (“infi nity”) rather than the arithmetically correct result 3.0e410, indicating that arithmetic overflow has occurred.

Arithmetic underflow

- a condition that occurs when a calculated result is too small in magnitude to be represented,
  >>>1.0e-300 / 1.0e100
  >>>0.0
- This results in 0.0 rather than the arithmetically correct result 1.0e-400, indicating that arithmetic underflow has occurred.

LET'S TRY IT

From the Python Shell, enter the following and observe the results.

```
>>> 1.2e200 * 2.4e100
???

>>> 1.2e200 * 2.4e200
???
```

```
>>> 1.2e200 / 2.4e100
???

>>> 1.2e-200 / 2.4e200
???
```

Arithmetic overflow occurs when a calculated result is too large in magnitude to be represented.

Arithmetic underflow occurs when a calculated result is too small in magnitude to be represented

Repeated Print

```
>>>print('a'*15)
# prints 'a' fifteen times

>>>print('\n'*15)
# prints new line character fifteen times
```

Complex Numbers

- A complex number consists of an ordered pair of real floating point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the complex number.
- **complex(x)** to convert x to a complex number with real part x and imaginary part zero
- **complex(x, y)** to convert x and y to a complex number with real part x and imaginary part y.
- x and y are numeric expressions

Complex Numbers

```
>>> 1j * 1J
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2 + 1j) * 3
(6+3j)
```

- A = 1+2j; B=3+2j
- # Mutpile statements can be given in same line using semicolon
- C = A+B; print(C)

Complex Numbers

prints real part of the number

- print(A.real)

prints imaginary part of the number

- print(A.imag)

Can do operations with part of complex number

- print(A.imag+3)

Input and output function

Input function : input

```
Basic_pay = input('Enter the Basic Pay: ')
```

Output function : print

```
print('Hello world!')
print(‘Net Salary', salary)
```

By Default…

- **Input function reads all values as strings**, to convert then to integers and float, use the function int() and float()

Type conversion…

```
line = input('How many credits do you have?')
num_credits = int(line)
line = input('What is your grade point average?')
gpa = float(line)
```

Here, the entered number of credits, say '24', is converted to the equivalent integer value, 24, before being assigned to variable num_credits. For input of the gpa, the entered value, say '3.2', is converted to the equivalent floating-point value, 3.2. Note that the program lines above could be combined as follows,

```
num_credits = int(input('How many credits do you have? '))
gpa = float(input('What is your grade point average? '))
```

Assignment Statement

Statement	Type
spam = 'Spam'	Basic form
spam, ham = 'yum', 'YUM'	Tuple assignment (positional)
[spam, ham] = ['yum', 'YUM']	List assignment (positional)
a, b, c, d = 'spam'	Sequence assignment, generalized
a, *b = 'spam'	Extended sequence unpacking (Python 3.X)
spam = ham = 'lunch'	Multiple-target assignment
spams += 42	Augmented assignment (equivalent to spams = spams + 42)

Range

```
>>>a,b,c = range(1,4)
>>>a
1
>>>b
2
```

```
>>> S = "spam" >>> S += "SPAM"  # Implied concatenation
 >>> S
'spamSPAM'
```

Assignment is more powerful in Python

```
>>> nudge = 1
>>> wink  = 2
>>> nudge, wink = wink, nudge
# Tuples: swaps values
# Like T = nudge; nudge = wink; wink = T
>>> nudge, wink
(2, 1)
```