Introduction to Python

Unit-4

Python is an interpreted high-level programming language for general-purpose programming. Created by ***Guido Van Rossum*** in 1989 parallely when ABC was under development. The short comings of ABC were overcome in Python. Its first release was in 1991, Python has been designed with a philosophy that emphasizes code readability, and a terse syntax that allows programmers to express concepts in fewer lines of code, particularly using significant whitespace.

Python features

- Python features a dynamic type system and automatic memory management.
- It supports
    - multiple programming paradigms,
    - object-oriented,
    - functional and
    - procedural
- Has a large and comprehensive standard library.

A dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, functional and procedural, and has a large and comprehensive standard library.

Python interpreters are available for many operating systems. **CPython**, is open source software and has a community-based development model. CPython is managed by the non-profit Python Software Foundation.

Why should you learn to write programs?

Writing programs (or programming) is a **very creative** and **rewarding activity**. Programs are written for many reasons, ranging from making ones living to solving a difficult data analysis problem or to have fun to help someone else solve a problem.

Computers are fast and have vast amounts of memory and could be very helpful to us if we knew the language to explain to the computer what we would like it to "do next". If we knew this language, we could tell the computer to do tasks on our behalf that were repetitive. Interestingly, the kinds of things computers can do best are often the kinds of things that we humans find boring and mind-numbing.

Skills required for a programmer:
• First, need to know the programming language (Python) - know the vocabulary and the grammar. To be able to spell the words in this new language properly and needs to know how to construct well-formed "sentences" in this new language.
• Second, "tell a story". In writing a story, words and sentences are combined to convey an idea to the reader. There is a skill and art in constructing the story, and skill in story writing is improved by doing some writing and getting some feedback. In programming, program is the "story" and the problem is to solve is the "idea".

Terminology: interpreter and compiler

Python is a high-level language intended to be simple for humans to read and write and for computers to read and process. Other high-level languages include Java, C++, PHP, Ruby, Basic, Perl, JavaScript, and many more. The actual hardware inside the Central Processing Unit (CPU) does not understand any of these high-level languages. The CPU understands a machine language. Machine language is very simple and very tiresome to write because it is represented all in zeros and ones.

Since machine language is tied to the computer hardware, machine language is not portable across different types of hardware. Programs written in high-level languages can be moved between different computers by using a different interpreter on the new machine or recompiling the code to create a machine language version of the program for the new machine.

These programming language translators fall into two general categories:

(1) interpreters and
(2) compilers.

An interpreter reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on the fly. Python is an interpreter and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it immediately and is ready for us to type another line of Python. Some of the lines of Python tell Python that you want it to remember some value for later. We need to pick a name for that value to be remembered and we can use that symbolic name to retrieve the value later.

Such variable are used to refer to the labels which in turn refer to this stored data.

For Ex.

```
>>> x = 8
>>> print(x)
8
>>> y = x * 10
>>> print(y)
80
```

In this example, Python remembers the value eight and use the label *x* so later when the value is retrieved later. We verify that Python has actually remembered the value using *print*. Then we ask Python to retrieve *x* and multiply it by ten and put the newly computed value in *y*. Then we ask to print out the value currently in *y*.

On the other hand a compiler needs to be given the entire program in a file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machine language into a file for later execution. On a Windows system, often these executable machine language programs have a suffix of ".exe" or ".dll" which stand for "executable" and "dynamic link library" respectively. On Linux and Macintosh, there is no suffix that uniquely marks a file as executable.

The Python interpreter is written in a high-level language called "C". The actual source code for the Python interpreter is available at **www.python.org** So Python is a program itself and it is compiled into machine code. When Python is installed on your computer (or the vendor

installed it), machine-code copy gets copied of the translated Python program onto your system. In Windows, the executable machine code for Python is inside a file with a name like:

C:\Python35\python.exe

Writing a program

Python interpreter can be used for solving not very complex problems. When we want to write a program, we use a text editor to write the Python instructions into a file, which is called a *script*. By convention, Python scripts have names that end with **.py**. To execute the script, Python interpreter is given the name of the file. In a Unix or Windows command window, type python **hello.py** as follows:

```
csev$ cat hello.py
      print('Hello world!')
csev$ python hello.py
      Hello world!
csev$
```

The "**csev$**" is the operating system prompt, and the "***cat hello.py***" shows that the file "**hello.py**" has a one-line Python program to print a string.

Variables, expressions, and statements

Values and types:

A *value* is one of the basic things a program works with, like a letter or a number. These values belong to different *types*: 2 is an integer, and "Hello, World!" is a *string*, because it contains a "string" of letters. Strings can be identified because they are enclosed in quotation marks.

```
python              ----python interpreter
>>> print(4)
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<class 'str'>

>>> type(17)
<class 'int'>

>>> type(3.2)
<class 'float'>
```

strings belong to the type **str** and integers belong to the type **int** and real numbers belong to **float**.

```
>>> type('17')
<class 'str'>
```

```
>>> type('3.2')
<class 'str'>
```

They're strings because they are enclosed in quotes.

When a large integer is typed with commas between groups of three digits, as in 1,000,000.

This is not a legal integer in Python, but it is legal number as it's not a syntax error :

```
>>> print(1,000,000)
1 0 0
```

Python interprets 1,000,000 as a comma separated sequence of integers, and prints it with spaces between.

Variables

A variable is an identifier name that refers to a value. An *assignment statement* creates new variable and gives it a value:

```
>>> message = 'Make hay while the sun shines'
>>> n = 21
>>> pi = 3.1415926535897931
```

To display the value of a variable, you can use a print statement:

```
>>> print(message)
Make hay while the sun shines
>>> print(n)
21
>>> print(pi)
3.141592653589793
```

The type of a variable is the type of the value it refers to.

```
>>> type(message)
    <class 'str'>
>>> type(n)
    <class 'int'>
>>> type(pi)
    <class 'float'>
```

Variable names and keywords

Rules to declare a Variable:

- Variable names can be arbitrarily long.
- They can contain both letters and numbers,
- They cannot start with a number or special characters.
- Uppercase letters are allowed.
- The underscore character (_) can appear in a name.
- Variable names can start with an underscore character.

Variable names starting with an underscore character are generally avoided unless it's been used for writing library code for others to use.

Ex. Illegal variable names:

```
>>> 76trombones = 'Neon Lights'
SyntaxError: invalid syntax

>>> more@ = 1000000
SyntaxError: invalid syntax

>>> class = 'Advanced Geometry of Space'
SyntaxError: invalid syntax
```

Python reserves 33 keywords:

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

Expressions

An *expression* is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

```
17
x
x + 17
```

If you type an expression in interactive mode, the interpreter *evaluates* it and displays the result:

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

But in a script, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

Exercise 1: Type the following statements in the Python interpreter to see what they do:
5
x = 5
x + 1

Order of operations

When more than one operator appears in an expression, the order of evaluation depends on the *rules of precedence*. For mathematical operators, Python follows mathematical convention. The acronym *PEMDAS* is a useful way to remember the rules:

- *P*arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8.
- *E*xponentiation has the next highest precedence, so 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.
- *M*ultiplication and *D*ivision have the same precedence, which is higher than *A*ddition and *S*ubtraction, which also have the same precedence. So 2*3-1 is 5, not 4, and 6+4/2 is 8.0, not 5.
- Operators with the same precedence are evaluated from left to right. So the expression 5-3-1 is 1, not 3, because the 5-3 happens first and then 1 is subtracted from 2.

When in doubt, always put parentheses in your expressions to make sure the computations are performed in the order you intend.

Modulus operator

The *modulus operator* works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

```
>>> quotient = 7 // 3
>>> print(quotient)
2
>>> remainder = 7 % 3
>>> print(remainder)
1
```

String operations

The + operator works with strings, but it is not addition in the mathematical sense. Instead it performs *concatenation*, which means joining the strings by linking them end to end.

For example:

```
>>> first = 10
>>> second = 15
>>> print(first+second)
25
>>> first = '100'
```

```
>>> second = '150'
>>> print(first + second)
100150
```

The output of this program is 100150.

Input/output functions:

Asking the user for input

Python provides a built-in function called ***input*** that gets input from the keyboard. When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and ***input*** returns what the user typed as a string.

```
>>> inp = input()
It's a great way
>>> print(inp)
```

print statement:

A statement is a unit of code that the Python interpreter can execute. **print** is an executable statement. When **print** statement is typed in interactive mode, the interpreter executes it and displays the result, if there is one.

Ex.1. A String can be passed to be displayed to the user before pausing for input:

```
>>> name = input('What is your name?\n')
What is your name?
Kiran
>>> print(name)
Kiran
```

Ex. 2. Two add two numbers by accepting from the user

```
num1=input('Enter the first number:');
num2=input('Enter the second number:');
sum =int(num1)+int(num2)
print('The sum is :' , sum)
```

Statements

A statement is a unit of code that the Python interpreter can execute. **print** being an expression statement and assignment. When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.

```
print(1)
x = 2
print(x)
```

produces the output
1
2

Operators and operands

Operators are special symbols that represent computations like addition and multiplication. The operators applied to values or variable are called *operands*. The operators +, -, *, /, and ** perform addition, subtraction, multiplication, division, and exponentiation, as in the following examples:

20+32 hour-1 hour*60+minute minute/60 5**2 (5+9)*(15-7)

There has been a change in the division operator between Python 2.x and Python 3.x. In Python 3.x, the result of this division is a floating point result:

>>> minute = 59
>>> minute/60
0.9833333333333333

The division operator in Python 2.0 would divide two integers and truncate the result to an integer:
>>> minute = 59
>>> minute/60
0

To obtain the same answer in Python 3.0 use floored (// integer) division.
>>> minute = 59
>>> minute//60
0

Conditional execution

Boolean expressions
A *boolean expression* is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

>>> 5 == 5
True
>>> 5 == 6
False
{}

True and False are special values that belong to the class bool; they are not strings:
>>> type(True)
<**class** 'bool'>
>>> type(False)
<**class** 'bool'>

The == operator is one of the *comparison operators*; the others are:

x != y *# x is not equal to y*
x > y *# x is greater than y*
x < y *# x is less than y*
x >= y *# x is greater than or equal to y*
x <= y *# x is less than or equal to y*
x is y *# x is the same as y*
x is not y *# x is not the same as y*

Logical operators

There are three *logical operators*: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English.
For example,
x > 0 and x < 10 is true only if x is greater than 0 *and* less than 10.

n%2 == 0 or n%3 == 0 is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y.

Comments

Programming reflects programmer's way of thinking in order to describe the single step's that were followed to solve a problem using a Computer. Commenting helps the programmers to explain the thought process involved, and helps the programmer and others to understand the intention of the code. This allows one to easily find errors, to fix them, to improve the code later on, and to reuse it in other applications as well.

Single line Comment:
Single line Comment(s) in Python are written starting with # (hash) character

```
# Program to read two numbers and
# display it to the screen

Y=25//2          # finds the integer division
```

MultiLine Comments

```
" " "
Write a Python Program to read two numbers and display the remainder to the screen after
dividing the first number by second.
" " "
```

Short-circuit evaluation of logical expressions

Python processes a logical expression such as x >= 2 and (x/y) > 2, by evaluating the expression from left to right. When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical stops because the overall value is already known, it is called *short-circuiting* the evaluation.

The short-circuit behavior leads to a clever technique called the ***guardian pattern***. Consider the following code sequence in the Python interpreter:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Debugging

Python displays an error message that contains a lot of information. The most useful parts are usually:

- What kind of error it was, and
- Where it occurred.

Syntax errors are usually easy to find, but there are few tricky errors such as ***Whitespace*** because spaces and tabs are invisible and most ignore them.

```
>>> x = 5
>>>  y = 6
File "<stdin>", line 1
y = 6
^
IndentationError: unexpected indent
```

In this example, the problem is that the second line is indented by one space. But the error message points to y, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

In general, error messages tell where the problem was discovered, but that is often not where it was caused.

Built-in functions

In the context of programming, a *function* is a set statements grouped under a name that performs a specific task. When a function is defined, first name is specified and then the set or sequence of statements. Function is called by the it's name.

```
>>> type(32)
<class 'int'>
```

The name of the function is **type**. The expression in parentheses is called the *argument* of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the **type** function, is the type of the argument. The result sent back by the function is called the *return value*.

The max and min functions give us the largest and smallest values in a list, respectively:

```
>>> print ( max('Hello world') )
        'w'
>>> print ( min('Hello world') )
        ' '
```

The **max** function tells us the "largest character" in the string (which turns out to be the letter "w") and the **min** function shows us the smallest character (which turns out to be a space).

A very common built-in function is the **len** function which tells how many items are in its argument. If the argument to **len** is a string, it returns the number of characters in the string.

```
>>> len('Hello world')
11
```

Type conversion functions

Python also provides built-in functions that convert values from one type to another. The **int** function takes any value and converts it to an integer, if it can, or displays an error otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int() with base 10: 'Hello'

>>> int(-2.3)
-2
```

Rounds by chopping the fractional part.

float converts integers and strings to floating-point numbers:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

Finally, str converts its argument to a string.

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

Random numbers

Given the same inputs, most computer programs generate the same outputs every time, so they are said to be *deterministic*. In determinism, it's expected that the same calculation yield's the same result. For some applications, though, the computer needs to be unpredictable. Games are an obvious example, but there are more.

The function **random** returns a random float between 0.0 and 1.0 (including 0.0 but not 1.0). Each time random is called, the next number is generated in a long series. The following program produces the following list of 10 random numbers between 0.0 and up to but not including 1.0:

```
import random
for i in range(10):
        x = random.random()
        print(x)
```

The function **randint** takes the parameters low and high, and returns an integer between low and high (including both).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

To choose an element from a sequence at random, you can use choice:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Math functions

Python has a math module that provides most of the familiar mathematical functions. Before using the module, it must be imported:

>>> import math
This statement creates a *module object* named math. If you print the module object, you get some information about it:
>>> print(math)
<module 'math' (built-in)>

The module object contains the functions and variables defined in the module. To access one of the functions, specify the name of the module and the name of the function, separated by a dot (also known as a period). This format is called ***dot notation***.

```
>>> ratio = signal_power/noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

The first example computes the logarithm base 10 of the signal-to-noise ratio. The math module also provides a function called log that computes **logarithms base e.** The second example finds the sine of radians. The name of the variable is a hint that **sin** and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

The expression **math.pi** gets the variable **pi** from the math module. The value of this variable is an approximation of **pi**, accurate to about 15 digits.