BE 101-05 : Introduction to Computing and Problem Solving

Chapter 8 - Python fundamentals

Narasimhan T.

1 Introduction

Python is a high-level, interpreted language. Its features are:

- **Python is interpreted**: Python is processed at runtime by the interpreter. You need not compile your program before executing it.
- **Python has simple, conventional syntax**: Python statements are very close to those of pseudocode algorithms, and Python expressions use the conventional notation found in algebra.
- **Python is highly interactive**: Expressions and statements can be entered at an interpreter's command prompts to allow the programmer to try out experimental code and receive immediate output.
- **Python is object-oriented**: Python supports object-oriented programming concepts.
- **Python scales well**: It is very easy for beginners to write simple programs in Python. Python also includes all of the advanced features of a modern programming language.

2 Getting started

Let us now get started with learning Python. There is a close analogy between learning English language and learning Python (or any other programming language). The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. And finally, an article consists of multiple paragraphs. This is shown in Figure 2.1.
Learning Python is similar and easier as shown in Figure 2.2. We first learn about alphabets, numbers and special symbols that are used in Python, then learn how using them; constants, variables and keywords are constructed. These are then combined to form an expression or instruction. A group of instructions would be combined later on to form a function. Program is a collection of different functions.

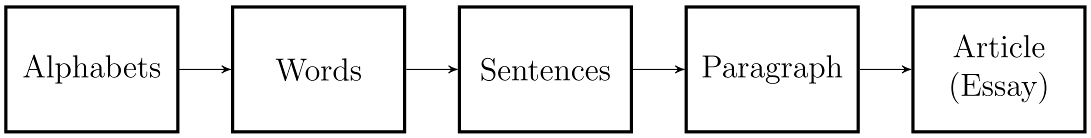

Figure 2.1: Steps in learning English

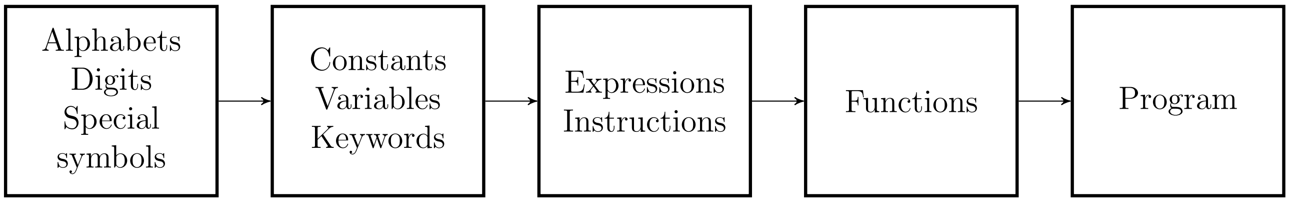
Alphabets Digits Special symbols
Constants Variables Keywords
Expressions Instructions
Functions
Program

Figure 2.2: Steps in learning Python

3 Character set

The set of characters supported by a programming language is called character set. A character denotes any alphabet, digit or special symbol used to represent information. Python supports the following characters:

- upper case alphabets (*A*–*Z*)
- lower case alphabets (*a*–*z*)
- digits (0–9)
- special symbols like @,#,%,$ etc.

4 Constants, Variables and Keywords

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords. A 'constant' is an entity whose value doesn't change. 3, 100 etc. are all constants. The data on which programs operate are stored in various memory locations. To make the retrieval and usage of the data values easy these memory locations are given names. Since the value stored in each location may change from time to time, the names given to these locations are called 'variable names' or simply 'variables'. Thus variable is a name that refers to a value. Programmers generally choose names for their variables that are meaningful. They can contain both letters and numbers. You need to be careful of a few rules when choosing names for your variables:

1. Must start with a letter or underscore (_).
2. Can be followed by any number of letters, digits, or underscores.
3. Python variable names are case sensitive; thus, the variable `WEIGHT` is a different name from the variable `weight`.
4. Cannot be a keyword. Keywords, also called **reserved words** are special words reserved for other purposes and thus cannot be used for variable names. Python has thirty three keywords as listed in Figure 4.1. All the keywords except `True, False` and `None` are in lower-case and they must be written as it is.

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

Figure 4.1: Reserved words in Python

5 Data types

The data stored in memory can be of different types. For example, your roll number is stored as a number but your name as string. Any variable has its associated data type. Python has various standard data types that are used to define the operations possible on them and how the values are stored in memory. The data types supported by Python are:

- Numbers
- String
- List
- Tuple
- Dictionary

5.1 Numbers

Number or numeric data type is used to store numeric values. Consider the statements

```
a = 10
c = 7.52
```

Here both `a` and `b` are of number data type. There are four distinct numeric types:

Type	Description	Examples
`int`	integers (numbers without decimal point)	`7,198`
`long`	long integers	`12L,25789L`
`float`	numbers with decimal point	`3.14,6.023`
`complex`	complex numbers	`3+4j, 10j`

The integers include `0`, positive whole numbers, and negative whole numbers. The most common implementation of Python's `int` data type consists of the integers from $-2,147,483,648(-2^{31})$ to $2,147,483,647(2^{31}-1)$. When the value of an integer exceeds these limits, Python automatically uses the `long` data type to represent it. A long integer looks just like a regular integer but can end with the letter `L`. The magnitude of a long integer can be quite large, but is still limited by the memory of the computer.
Python uses floating-point numbers to represent real numbers. The values of float type range from approximately -10^{308} to 10^{308} and have 16 digits of precision (number of digits after the

decimal point). A floating-point number can be written using either ordinary decimal notation or scientific notation. Scientific notation is often useful for mentioning very large numbers. See the examples below:

Decimal notation	**Scientific notation**	**Meaning**
`3.146`	`3.146e0`	3.146×10^{0}
`314.6`	`3.146e2`	3.146×10^{2}
`0.3146`	`3.146e-1`	3.146×10^{-1}
`0.003146`	`3.146e-3`	3.146×10^{-3}

Complex numbers are written in the form, `x+yj`, where `x` is the real part and `y` is the imaginary part.
`int` type has a subtype `bool`. Any variable of type `bool` can take one of the two possible boolean values `True` and `False`.

Python 2 vs Python 3

There are actually two flavours of Python:

- Python 2 – latest release is Python 2.7.12 (as on 15/10/16)
- Python 3 – latest release is Python 3.5.2 (as on 15/10/16)

Python 2 had separate types for `int` and `long`. Python 3 has just one integer type, `int` which behaves mostly like the old `long` type from Python 2.

5.2 Strings

String is sequence of characters. We can use a pair of single quotes or double quotes to represent strings. `"Hi", "8.5", ‘hello’` are all strings. Multi-line strings can be denoted using triple quotes, `‘‘‘` or `"""`. The following is an example of multi-line string:

```
‘‘‘a multiline
string’’’
```

We pause the discussion of data types here. The remaining data types : lists, tuples and dictionaries will be explored later.

6 Statements

A statement is an instruction that the Python interpreter can execute. A statement could be an expression statement or a control statement. An expression statement contains an expression which is evaluated by the interpreter. Expressions provide an easy way to perform operations on data values to produce other data values. A control statement is used to represent advanced features of a language like decision making and looping.

7 Arithmetic Expressions

An arithmetic expression consists of operands and operators combined in a meaningful manner. Operands are data items on which various operations are performed. The operations are denoted by special symbols called operators. The operands could be constants or variables. When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.

7.1 Types of Operators

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

7.1.1 Arithmetic Operators

The various arithmetic operators in Python are tabulated in Table 7.1. All the operators except floor division are familiar to you. Floor division is also called integer division. The statement

```
a=7.0 / 2
```

will result in `a=3.5`, whereas the statement

```
a=7.0 // 2
```

will yield `a=3.0`.

Operation	Operator
Negation	-
Addition	+
Subtraction	−
Multiplication	*
Division	/
Floor division	//
Remainder	%
Exponentiation	**

Table 7.1: Arithmetic operators

Operation	Operator
Equal to	==
Not equal to	!=
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=

Table 7.2: Comparison operators

7.1.2 Comparison Operators

Table 7.2 shows the various comparison operators. The result of a comparison is a Boolean value either `True` or `False`. For example, if `x = 5` and `y = 3`, then the comparison `x > y` will result in `True`.

7.1.3 Assignment operator

The assignment operator used in Python is '='. The **assignment statement** creates new variables and gives them values. For example, the assignment statement `n = 17` creates a new variable whose value is 17. The statement `a = b + 5` first evaluates `b + 5` and then assigns the result to `a`. Python allows you to assign a single value to several variables simultaneously. For example consider

```
a = b = 5
```

Here both variables `a` and `b` are assigned the value 5. You can also assign multiple objects to multiple variables. For example, the statement

```
a, b, c = 1, 2.5, "ram"
```

assigns `1` to `a`, `2.5` to `b` and `ram` to `c`. Python also allows a shorthand notation in writing assignment statements. The statement

```
a += c
```

is equivalent to writing

```
a = a + c
```

Any arithmetic operator can be used in this shorthand notation.

7.1.4 Logical Operators

Python includes three Boolean or logical operators, `and` , `or` , and `not` . Both the `and` operator and the `or` operator expect two operands and are hence called binary operators. The `and` operator returns `True` if and only if both of its operands are `True`, and returns `False` otherwise. The `or` operator returns `False` if and only if both of its operands are `False`, and returns `True` otherwise. The `not` operator expects a single operand and is hence called unary operator. It returns the logical negation of the operand, that is, `True` , if the operand is `False`, and `False` if the operand is `True`.
The behaviour of each operator can be completely specified in a **truth table** for that operator. The first row in a truth table contains labels for the operands and the expression being computed. Each row below the first one contains one possible combination of values for the operands and the value resulting from applying the operator to them. Tables 7.3,7.4 and 7.5 show the truth tables for `and` , `or` , and `not` .
In the context of logical operators, Python interprets all non zero values as `True` and zero as `False`. See the following examples.

Expression	Interpretation	Value
`7 and 1`	`True and True`	`7`
`-2 or 0`	`True or False`	`-2`
`-100 and 0`	`True and False`	`0`

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

Table 7.3: Truth table for logical AND

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

Table 7.4: Truth table for logical OR

a	not a
False	True
True	False

Table 7.5: Truth table for logical NOT

7.1.5 Bitwise operators

Bitwise operators take the binary representation of the operands and work on their individual bits, one bit at a time.

b_1	b_2	$b_1 \,\&\, b_2$
0	0	0
0	1	0
1	0	0
1	1	1

Table 7.6: Truth table for bitwise AND

b_1	b_2	$b_1 \mid b_2$
0	0	0
0	1	1
1	0	1
1	1	1

Table 7.7: Truth table for bitwise OR

b_1	b_2	$b_1 \wedge b_2$
0	0	0
0	1	1
1	0	1
1	1	0

Table 7.8: Truth table for bitwise XOR

Bitwise AND

Bitwise AND is denoted by the symbol `&`. Its truth table is shown in Table 7.6. Suppose we want to perform `14 & 20`. We represent both operands in 8-bit binary representation. The operation of the bitwise AND on the numbers 14 (00001110 in binary) and 20 (00010100 in binary) is shown below.

$$\begin{array}{l} 00001110\ \& \\ 00010100 \\ \hline 00000100 \end{array}$$

The result is 4 (00000100 in binary). Thus `14 & 20 = 4`.

Bitwise OR

Bitwise OR is denoted by the symbol |. The truth table is shown in Table 7.7. Suppose we want to perform `7 | 25`. We represent both operands in 8-bit binary representation. The operation of the bitwise OR on the numbers 7 (00000111 in binary) and 25 (00011001 in binary) is shown below.

$$\begin{array}{l} 00000111\ \mid \\ 00011001 \\ \hline 00011111 \end{array}$$

The result is 31 (00011111 in binary). Thus `7 | 25 = 31`.

Bitwise XOR

Bitwise XOR is denoted by the symbol ^. Its truth table is shown in Table 7.8. Suppose we want to perform `10 ^ 18`. We represent both operands in 8-bit binary representation. The operation of the bitwise XOR on the numbers 10 (00001010 in binary) and 18 (00010010 in binary) is shown below.

$$\begin{array}{l} 00001010\ \hat{} \\ 00010010 \\ \hline 00011000 \end{array}$$

The result is 24 (00011000 in binary). Thus `10 ^ 18 = 24`.

One's complement

One's complement is denoted by the symbol ~. One's complement of a number is found by changing all the zeroes to ones and ones to zeroes in the binary representation of the number. Its truth table is shown below.

b_1	$\sim b_1$
0	1
1	0

Suppose we want to determine `~9`. The binary of 9 is 1001. Changing 0's to 1's and 1's to 0's, we get $0110 = 6$. Thus `~9 = 6`.

Binary left shift

Left shift operation is denoted by the symbol $\ll$. The expression `x` $\ll$ `n` shifts each bit of the binary representation of `x` to the left, `n` times. Each time we shift the bits left, the vacant bit position at the right end is filled with a zero.
Suppose we want to determine `15` $\ll$ `2`. We follow the steps below:

1. Write `15` in 8-bit binary representation. `15 = 00001111`

2. Left shifting the bits once, we get `00011110`

3. Performing left shift second time yields `00111100`

Thus `15` $\ll$ `2 = 00111100 = 60`. Left shifting a binary number n times is equivalent to multiplying with 2^n.

Binary right shift

Right shift operation is denoted by the symbol $\gg$. The expression `x` $\gg$ `n` shifts each bit of the binary representation of `x` to the right, `n` times. Each time we shift the bits right, the vacant bit position at the left end is filled with a zero.
Suppose we want to determine `15` $\gg$ `2`. We proceed as follows:

1. Write `15` in 8-bit binary representation. `15 = 00001111`

2. Right shifting the bits once, we get `00000111`

3. Performing right shift second time yields `00000011`

Thus `15` $\ll$ `2 = 0011 = 3`. Right shifting a binary number n times is equivalent to dividing by 2^n.

Operators	Meaning
()	Parentheses
**	Exponent
+, −, ∼	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, −	Addition, Subtraction
≪, ≫	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, ! =, >, >=, <, <=, `is`, `is not`, `in`, `not in`	Comparisions, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR

Table 7.9: Precedence rules

7.1.6 Membership Operators

These operators test for membership in a sequence such as strings. There are two membership operators that are used in Python.

- **in** – Evaluates to `True` if it finds a variable in the specified sequence and `False` otherwise.
- **not in** – Evaluates to `True` if it does not find a variable in the specified sequence and `False` otherwise.

We will discuss more about these operators when we discuss lists and tuples.

7.1.7 Identity Operators

is and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located in the same part of the memory. More about these operators will be dealt with, when we discuss about objects.

7.2 Precedence and associativity of operators

When an expression contains more than one operator, in what order the operations are to be performed? For this, there are rules of precedence in Python. These rules guide the order in which the various operations are carried out. For example, multiplication has precedence over addition. This means, if an expression has both addition and multiplication, addition will be performed only after multiplication. Thus operator precedence dictates how an expression is evaluated. The precedence of operators in Python are shown in Table 7.9. The operators are listed in groups in descending order – upper group has higher precedence than the lower ones. We find that in Table 7.9, more than one operator exist in the same group. These operators have the same precedence. If an expression has multiple operators with same precedence, the tie is resolved using rules of associativity. Almost all operators have left-to-right associativity. For example, multiplication and division have the same precedence. Hence, if both of them are present in an expression, left one is evaluated first. Exponentiation and assignment operations are right associative. The following examples should make the concepts clear.

Expression	Evaluation	Value
`3 + 4 * 2`	`3 + 8`	`11`
`(3 + 4 ) * 2`	`7 * 2`	`14`
`2 ** 3 ** 2`	`2 ** 9`	`512`
`(2 ** 3) ** 2`	`8 ** 2`	`64`
`-3 ** 2`	`-(3 ** 2)`	`-9`
`-(3) ** 2`	`(-3) ** 2`	`9`
`45 / 0`	`Error:cannot divide by 0`	
`5 % 0`	`Error:cannot divide by 0`	
`not True and False or True`	`(False and False) or True`	`True`

Some operators like comparison operators do not have associativity in Python. There are separate rules for sequences of this kind of operator and cannot be expressed as associativity.

8 Data input and output

To display output on the screen, Python uses the method `print`. It just prints whatever is given inside the quotes. For example, the statement

```
print "Hello World!"
```

will print the output

```
Hello World!
```

Consider the statements

```
a = 7
print "The value of a is",a
```

will print

```
The value of a is 7
```

Python prints as such, whatever is enclosed in `" "`. In the above example, the first occurrence of `a` is within double quotes. So it is printed as such. The second `a` occurs outside the quotes. Thus its value `7` gets printed.

To receive input from the user, Python provides two methods `raw_input()` and `input()`. `raw_input()` accepts the input as string. For example, the statements

```
yourName = raw_input("What is your name?")
print yourName
```

prompts the user to enter his name by displaying the *prompt message* `"What is your name?"`. The name entered by the user is assigned to the variable `yourName`. Then the name is displayed.

If you want give numbers as input, we use the `input()` method. Here is an example:

```
num = input("Enter a number")
print "The number you entered is", num
```

`input()` accepts the input as numbers; so any arithmetic operation can be performed on the received input.

9 Comments

As programs get bigger and more complicated, they get more difficult to read and understand. For this reason, it is a good idea to add notes to your programs to explain in English what the program is doing. These notes are called **comments**, and they are marked with the `#` symbol:

```
# The line below prints Hello
print "Hello"
```

In this case, the comment appears on a separate line. You can also put comments at the end of a line:

```
print "Hello" # This line prints Hello
```

Everything from the `#` to the end of the line is ignored by the interpreter while execution–it has no effect on the program. The message is intended for the programmer himself or for others who might use this code. Usually we include comments to explain the working of complex or tricky sections of code. Python also supports comments that extend multiple lines, one way of doing it is to use `#` in the beginning of each line. Here is an example:

```
# This is a long comment
# and it extends
# to multiple lines
```

10 Programming examples

Program 1. To print “My first Python program”.

```
print "My first Python program"
```

Program 2. To input the user’s name and print a greeting message.

```
name=raw_input("Enter your name")
print "Hello",name
```

Program 3. To input a number and display it.

```
num=input("Enter a number")
print "The number you entered is",num
```

Program 4. To add and subtract two input numbers.

```
a=input("Enter the first number")
b=input("Enter the second number")
sum=a+b
```

```
difference=a-b
print "The sum is",sum
print "The difference is",difference
```

Program 5. To input the sides of a rectangle and find its perimeter.

```
length=input("Enter the length of the rectangle")
breadth=input("Enter the breadth of the rectangle")
perimeter=2*(length+breadth)
print "Perimeter of the rectangle is",perimeter
```

Program 6. To input the side of a square and find its area.

```
side=input("Enter the side of the square")
area=side**2
print "Area of the square is",area
```

Program 7. To input the radius of a circle and find its circumference.

```
import math
radius=input("Enter the radius")
c=2*math.pi*radius
print "Circumference of the circle is",c
```

In the code above, instead of using the value of π directly in the program, we let the Python interpreter determine its value accurately. For this we write, `math.pi`. Note that when we use `math` in the program, we include the statement `import math` as the first line.

Program 8. To input two values a and b and then find a^b.

```
import math
a=input("Enter the base")
b=input("Enter the exponent")
c=math.pow(a,b)
print a,"to the power",b,"is",c
```

Here `pow(a,b)` finds a^b. Again we include `import math` as the first line.

Program 9. To input two values a and b and then swap them.

```
a=input("Enter a number")
b=input("Enter another number")
print "The numbers before swapping are a =",a,"and b =",b
temp=a
a=b
b=temp
print "The numbers after swapping are a =",a,"and b =",b
```

Program 10. To input two values a and b and then swap them without using a third temporary variable

```
a=input("Enter a number")
b=input("Enter another number")
print "The numbers before swapping are a =",a,"and b =",b
a=a+b
b=a-b
a=a-b
print "The numbers after swapping are a =",a,"and b =",b
```

Program 11. Predict the output of the following code.

```
i=2
b=True
s="Hello"
f=7.18
l=12L
print type(i)
print type(b)
print type(s)
print type(f)
print type(l)
```

Solution: The `type` method gives the data type of a variable. Here is the output:

```
<type 'int'>
<type 'bool'>
<type 'str'>
<type 'float'>
<type 'long'>
```