Types, Operators, and Expressions

Types

In general, the structure of a Python program is as follows:

- Programs are composed of modules.
- Modules contain statements.
- Statements contain expressions.
- Expressions create and process objects.

In Python, everything is an object. Including values. Even simple numbers qualify, with values (e.g., 99), and supported operations (addition, subtraction, and so on). In Python, data takes the form of objects—either built-in objects that Python provides, or objects we create using Python classes or external language tools such as C extension libraries.

Following table shows fundamental Python's built-in object types and some of the syntax used to code their literals—that is, the expressions that generate these objects:

Object Type	**Examples**
Numbers	12, 2.67, 6+8j, 0b1011
Strings	'hai', "hello", "Python's Features", str('Python')
Lists	[1,2,3], [1,2,'three'], list(range(10)), list('hai')
Tuples	(1,2,3), (1,2,'three'), tuple(range(10)), tuple('hai')
Sets	{1,2,3}, set('hai')
Dictionaries	{'Mon':1, 'Tue':2, 'Wed':3}, dict(hours=10)

Following are some more types available in Python:

Object Type	**Examples**
Files	open('abc.txt')
Functions	def, lambda
Modules	import, __module__
Classes	objects, types, metaclasses
None	None
Booleans	True, False

Numbers

Numbers include the following:

- integers
- floating-point numbers
- complex numbers
- decimals with fixed precision
- rational numbers with numerator and denominator
- sets

Using third party extensions we have more types like matrices, vectors, etc.

Following are examples of numeric literals and constructors:

Literal	**Description**
12, -887, 0, 99999999999999999999998	Integers (unlimited size)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Floating-point numbers
0o177, 0x9ff, 0b101010	Octal, hex, and binary literals
6+4j, 2.0+9.0j, 7J	Complex number literals
Decimal('13.0'), Fraction(4, 7)	Decimal and fraction extension types
bool(X), True, False	Boolean type and constants

Floating-point numbers are implemented as C doubles in standard Cpython. The functions *hex*, *oct*, and *bin* can be used to convert integers to hexadecimal, octal, and binary formats respectively.

Complex numbers are internally implemented as pairs of floating-point numbers. Complex numbers can be ended with j or J. Complex numbers can also be created with *complex(real, imag)* function call.

We can use built-in functions like: *pow*, *abs*, *round*, *int*, *hex*, *bin*, etc on numbers. We can also use utility modules like *random*, *math* as follows:

```
>>> import random
>>> random.random()
0.865758460309634
>>> random.choice(range(100))
96

>>> import math
>>> math.sqrt(7)
2.6457513110645907
```

Decimals

Decimals are fixed precision floating point numbers. Decimals can be precise up to n digits after the decimal point. For example, 0.1+0.1+0.1-0.3 gives 5.551115123125783e-17. In such cases, we can use decimal as follows:

```
>>> from decimal import Decimal
>>> x=Decimal('0.1')
>>> y=Decimal('0.3')
>>> x+x+x-y
```

```
Decimal('0.0')
```

Default precision for decimal is 28 digits. We can set the precision as follows:

```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.14285714285714285714285714291')

#Set precision to 4 digits
>>> decimal.getcontext().prec = 4
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
```

Fraction

Fraction objects are used to implement rational numbers. It keeps both numerator and denominator explicitly. Following are examples for working with fractions in Python:

```
>>> from fractions import Fraction
>>> x=Fraction(2,7)
>>> y=Fraction(3,7)
>>> x+y
Fraction(5, 7)
>>> x*y
Fraction(6, 49)
>>> print(x)
2/7
```

Strings

A string is a sequence of one character strings. Strings are used to store textual information. Examples of strings:

‘Python’
‘Ramesh Kumar’
“A”
“123”

Sequence Operations on Strings

We can find the length (number of characters) of a string using the pre-defined function *len*:

```
>>>s = “hello”
>>>len(s)
5
```

We can also fetch individual characters from the strings using indexing. The index of first character starts from 0, next is 1, and so on:

```
>>>s = “hello”
>>>s[0]
‘h’
```

Negative indexes are used to index from right to left in a string:

```
>>>s = “Python”
>>>s[-1]
‘n’
```

Sequences like strings also support another form of indexing called slicing which is used to extract a part of the string:

```
>>>s = “Python”
>>>s[1:3]
‘yt’
```

More examples on slicing:

```
>>>s = “Python”
>>>s[2:]
thon
>>>s[:4]
‘Pyth’
>>>s[:len(s)]
‘Python’
>>>s[:-1]
‘Pytho’  #Everything except last character
>>>s[:]
‘Python’
```

Strings can be concatenated using the '+' operators as shown below:

```
>>>s = “Python”
>>>s + “ Rocks”
‘Python Rocks’
```

A string can be printed multiple times by using the power (**) operator as shown below:

```
>>>s = “Yo ”
>>>s**5
‘Yo Yo Yo Yo Yo ’
```

String Immutability

In Python, every object can be classified as mutable or immutable. Numbers, strings, and tuples are immutable, i.e., once assigned, values cannot be changed. Lists, dictionaries, and sets are mutable. For example, following will give error:

```
>>>s = “Python”
>>>s[1] = ‘a’
...
TypeError: 'str' object does not support item assignment
```

String Specific Methods

The *find* method can be used to find position of a given substring in the string:

```
>>>s="Python"
>>>s.find('Py')
0
>>>s.find('ab')
-1
```

The *replace* method can be used to replace a substring with a given string:

```
>>>s="Python"
>>>s.replace('Py', 'Go')
'Gothon'
>>>s.replace('ab', 'Go')
'Python'
```

The *split* method can be used to split a line into words based on the given delimiter. Default delimiter is space:

```
>>>line='this is a line'
>>>line.split()
['this', 'is', 'a', 'line']

>>>line='this,is,a,line'
>>>line.split(',')
['this', 'is', 'a', 'line']
```

The *upper* and *lower* methods can be used to turn a string into upper or lower case respectively:

```
>>>s='awesome'
>>>s=s.upper()
>>>s
'AWESOME'
>>>s=s.lower()
>>>s
'awesome'
```

The *isalpha* and *isdigit* methods can be used to find whether a string is containing all alphabets or digits respectively:

```
>>> s='alpha'
>>> s.isalpha()
True
>>> s.isdigit()
False
>>> s='123'
>>> s.isdigit()
True
```

The *rstrip* method can be used to remove white spaces at the right end of the string:

```
>>> line='this is a line    '
>>> line.rstrip()
'this is a line'
>>> line
'this is a line    '
```

We can substitute values in a string as follows:

```
>>> lang='Python'
>>> ver='3.x'
>>> 'Welcome to %s. Version %s.' % (lang,ver)
'Welcome to Python. Version 3.x.'

>>> 'Welcome to {0}. Version {1}.'.format(lang,ver)
'Welcome to Python. Version 3.x.'

>>> 'Welcome to {}. Version {}.'.format(lang,ver)
'Welcome to Python. Version 3.x.'
```

Booleans

Python provides Boolean data type called *bool* with values *True* and *False*. *True* represents integer 1 and *False* represents integer 0. *bool* is a sub class of integer class. Following are examples on Booleans:

```
>>> True == 1
True

>>> True is 1
False

>>> True and False
False

>>> True + 10
11

>>> False * 9
0
```

Operators

Following are different types of operators in Python:

- Arithmetic operators
- Relational operators
- Assignment operators
- Logical operators
- Bitwise operators
- Membership operators
- Identity operators

Arithmetic Operators

Following are various arithmetic operators available in Python:

Operator	Description	Example

+	Add two operands	x+y = 26
-	Subtract one operand from another	x-y = 14
*	Multiply one operand with another	x*y = 120
/	Divide one operand with another	x/y = 3.333
%	Remainder of division between two operands	x%y = 2
**	x to the power of y	x**y = 64000000
//	Floor division. Removes decimal part after division.	x//y = 3

Note: In above examples, x = 20 and y = 6

Relational Operators

Following are various relational operators available in Python:

Operator	Description	Example
= =	Returns true if values of both operands are equal . Otherwise false.	x==y is false
!=	Returns true if values of both operands are not equal. Otherwise false.	x!=y is true
< >	Returns true if values of both operands are not equal. Otherwise false.	x!=y is true
>	Returns true if left operand is greater than the right operand. Otherwise false.	x>y is true
<	Returns true if left operand is less than the right operand. Otherwise false.	x<y is false
>=	Returns true if left operand is greater than or equal to the right operand. Otherwise false.	x>=y is true
<=	Returns true if left operand is less than or equal to the right operand. Otherwise false.	x<=y is false

Note: In above examples, x = 20 and y = 6

Assignment Operators

Following are various assignment operators available in Python:

Operator	Description	Example
=	Assigns value of right side operand to left side operand	z=x; z=20
+=	Adds operand on left side with operand on right side and assigns the value to left side operand.	z+=x; z=20

-=	Subtracts operand on right side with operand on left side and assigns the value to left side operand.	z-=x; z=-20
*=	Multiplies operand on left side with operand on right side and assigns the value to left side operand.	z*=x; z=0
/=	Divide operand on left side with operand on right side and assigns the value to left side operand.	z/=x; z=0.0
%=	Assigns remainder of division to left side operand.	z%=x; z=0
**=	Assigns left operand power right operand to left side operand.	z**=x; z=0
//=	Assigns result of floor division to left side operand.	z//=x; z=0

Note: In above examples, x = 20 , y = 6, and z = 0

Bitwise Operators

Following are various bitwise operators available in Python:

Operator	Description	Example
& (and)	Performs bit-wise and of both operands	x&y = 1
| (or)	Performs bit-wise or of both operands	x|y = 5
^ (ex-or)	Performs exclusive-or of both operands	x^y = 4
~	Performs 1's complement of the operand	~x = -6
<<	Performs left shift of the left operand by n number of times	x<<2 = 20
>>	Performs right shift of the left operand by n number of times	x>>1 = 2

Note: In above examples, x = 5 , y = 1

Logical Operators

Following are various logical operators available in Python:

Operator	Description	Example
and	If both left side and right side expressions are true, it returns true. Otherwise, false.	True and True is True
or	If either or both of left side and right side expressions are true, it returns true. Otherwise, false.	True or False is True
not	If expression evaluates to true, it returns false or if the expression evaluates to false, it returns true.	not True is False

Membership Operators

Following are the membership operators available in Python:

Operator	Description	Example
in	Evaluates to true if it finds a variable or value in the given sequence. Otherwise, it returns true.	y in x is True
not in	Evaluates to true if it doesn't find a variable or value in the given sequence. Otherwise, it returns false.	y not in x is False

Note: In above examples, x = [1,2,3,4,5,6] , y = 2

Identity Operators

Following are the identity operators available in Python:

Operator	Description	Example
is	Evaluates to true if both the variables point to the same object in memory.	x is y returns True
is not	Evaluates to true if both the variables does not point to the same object in memory	x is not y returns False

Note: In above examples, x = 10, y = x

Expression Evaluation

A Python program contains one or more statements. A statement contains zero or more expressions. Python executes a statement by evaluating its expressions to values one by one. Python evaluates an expression by evaluating the sub-expressions and substituting their values.

Literal Expressions

A literal expression evaluates to the value it represents. Following are some examples of literal expressions:

10 => 10
'Welcome to Python' => 'Welcome to Python'
7.89 => 7.89
True => True

Binary Expressions

A binary expression consists of a binary operator applied to two operand expressions. Examples:

2*6 => 12
8-6 => 2
8==8 => True
1000 > 100 => True

‘hello’ + ‘world’ => ‘helloworld’

Unary Expressions

An unary expression contains one operator and single operand. Examples:

-(5/5) => -1
-(3*4) => -12
-(2**4) => -16
-10 => -10

Compound Expressions

In a binary or unary expression, if an operand itself is an expression, such expression is known as a compound expression. Examples:

3 * 2 + 1 => 7
2 + 6 * 2 => 14
(2 + 6) * 2 => 16

Variable Access Expressions

A variable access expressions allows us to access the value of a variable. Examples:

```
>>>x = 10
>>>(x + 2) * 4
48
>>>x
10
```

Control Statements

Generally, a Python script executes in a sequential manner. If a set of statements should be skipped or repeated again, we should alter the flow of control. The statements which allow us to alter the flow of control are known a control statements. Python supports the following control statements:

- if
- if...else
- elif ladder
- while
- for
- break
- continue

if Statement

A *if* statement can be used as a one way decision making statement. The syntax of *if* statement is as follows:

if condition:

```
    statement1
    statement2
    ....
    statementN
```

if...else Statement

A *if...else* statement can be used as a two way decision making statement. The syntax of *if...else* statement is as follows:

```
if condition:
    statement1
    statement2
    ....
    statementN
else:
    statement1
    statement2
    ....
    statementN
```

if...else Ternary Expression

Python supports *if...else* ternary expression. Its syntax is as follows:

```
expr1 if condition else expr2
```

Following is an example which demonstrates *if...else* ternary expression:

```
a=10
b=5
c = a if a>b else b
print(c)
```

while Loop

A *while* loop can be used to repeat a set of statements based on a condition. The syntax of *while* loop is as follows:

```
while test:
    statements
else:   #optional
    statements
```

As mentioned above, *else* part is optional. The *else* part executes only when *break* is not used.

for Loop

A *for* loop can be used to repeat a set of statements. The syntax of *for* loop is as follows:

```
for target in object:
```

```
        statements
else:   #optional
        statements
```

The *object* in the above syntax should be a collection of values like a list, string, tuple, etc. Like *while* loop, the *else* part is optional and executes only when *break* is not used.

break and continue

Both *break* and *continue* and jump statements which are used inside *while* loop or *for* loop. When *break* is used inside the loop, the control moves to the statement outside the enclosing loop. When *continue* is used inside the loop, the control moves to the first statement of the enclosing loop skipping all the remaining statements after the *continue* statement inside the enclosing loop.