Session 2

VARIABLE TYPE IN PYTHON

RAVINDRA KUDACHE

VARIABLE TYPES

- **Assigning Values to Variables:**

E:g

```
counter = 100 # An integer assignment
miles = 1000.0 # A floating point
name = "John" # A string
```

VARIABLE TYPES

Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example:

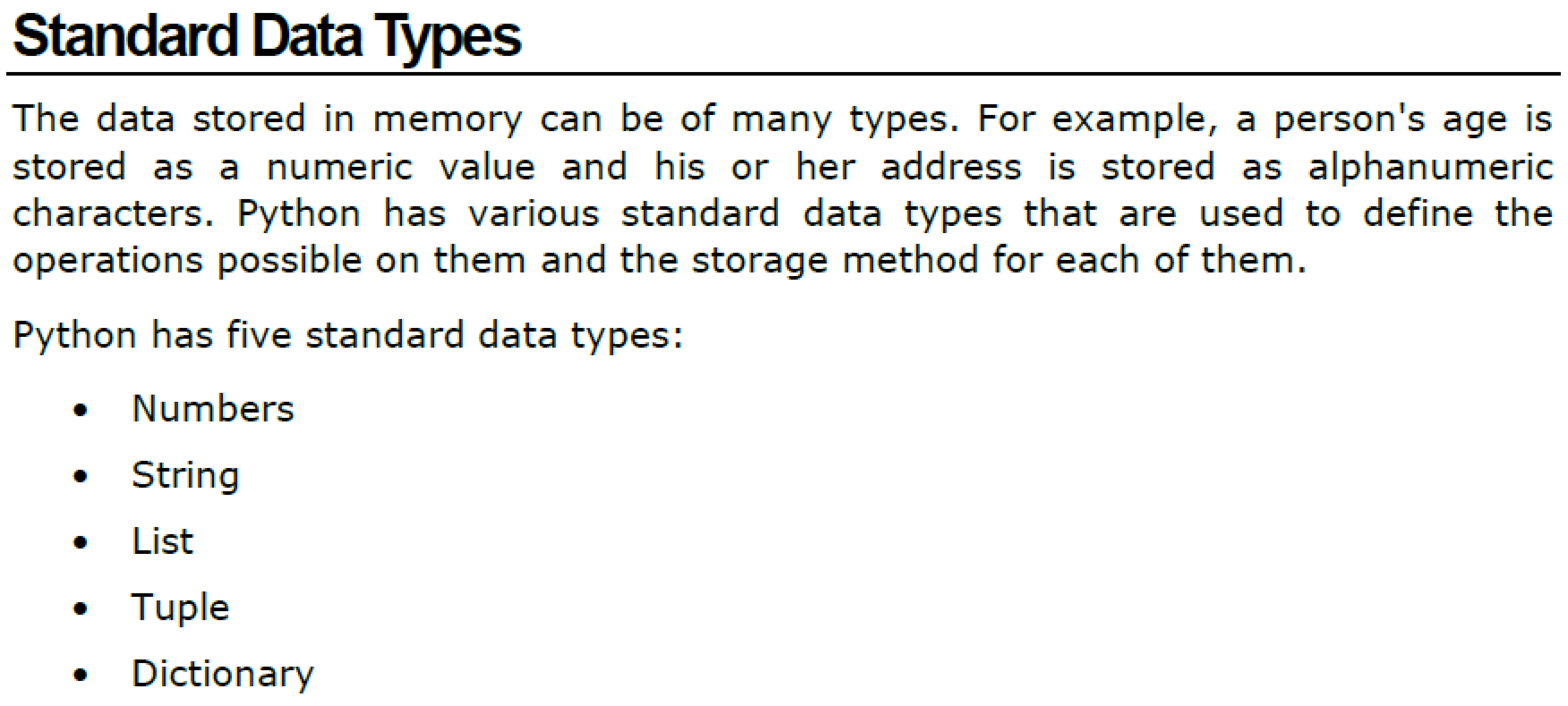
```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example:

```
a, b, c = 1, 2, "john"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

VARIABLE TYPES



Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types:

- Numbers
- String
- List
- Tuple
- Dictionary

VARIABLE TYPES

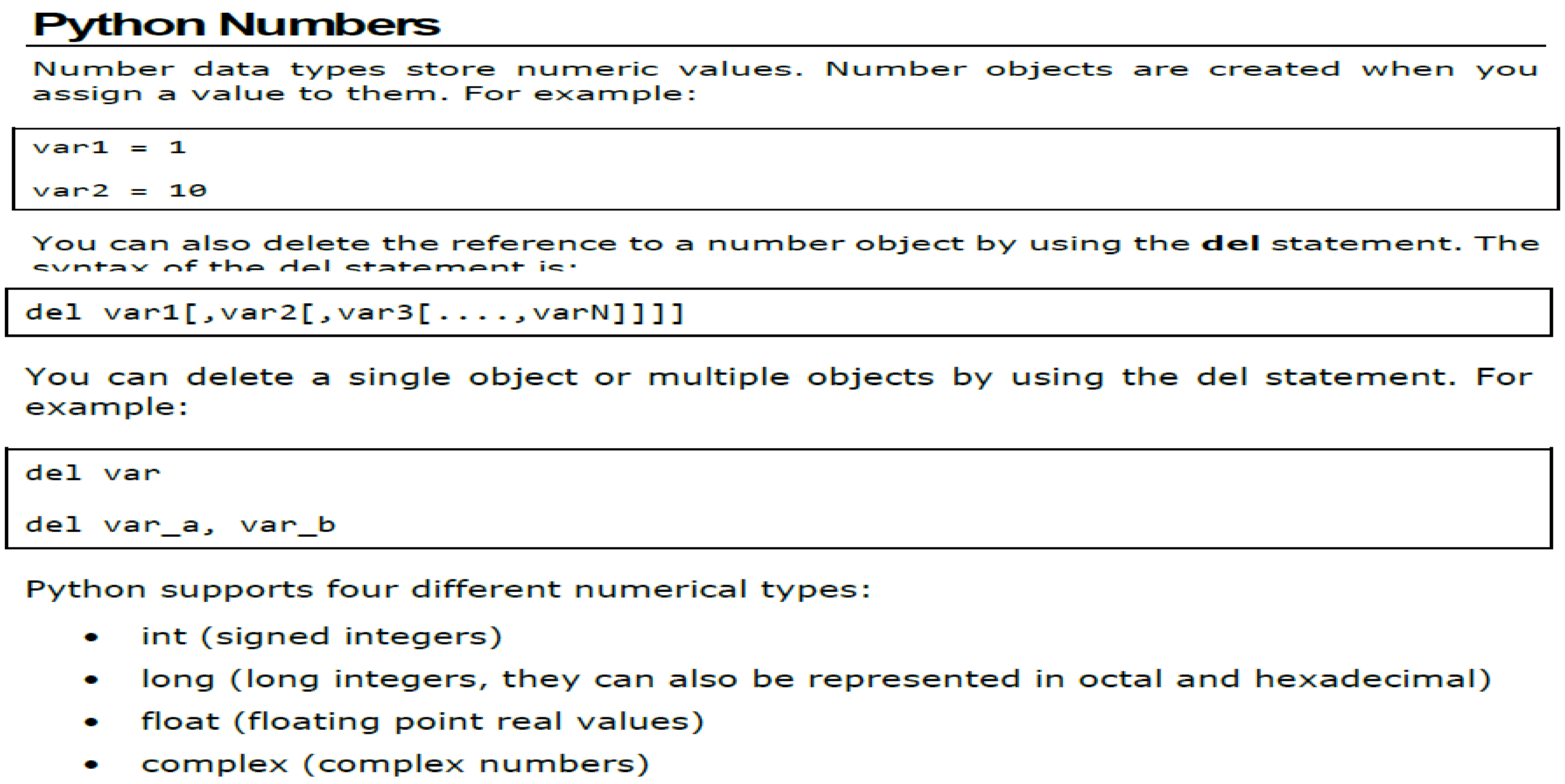
Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example:

```
var1 = 1

var2 = 10
```

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is:

VARIABLE TYPES



Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example:

```
var1 = 1

var2 = 10
```

You can also delete the reference to a number object by using the **del** statement. The syntax of the del statement is:

```
del var1[,var2[,var3[....,varN]]]]
```

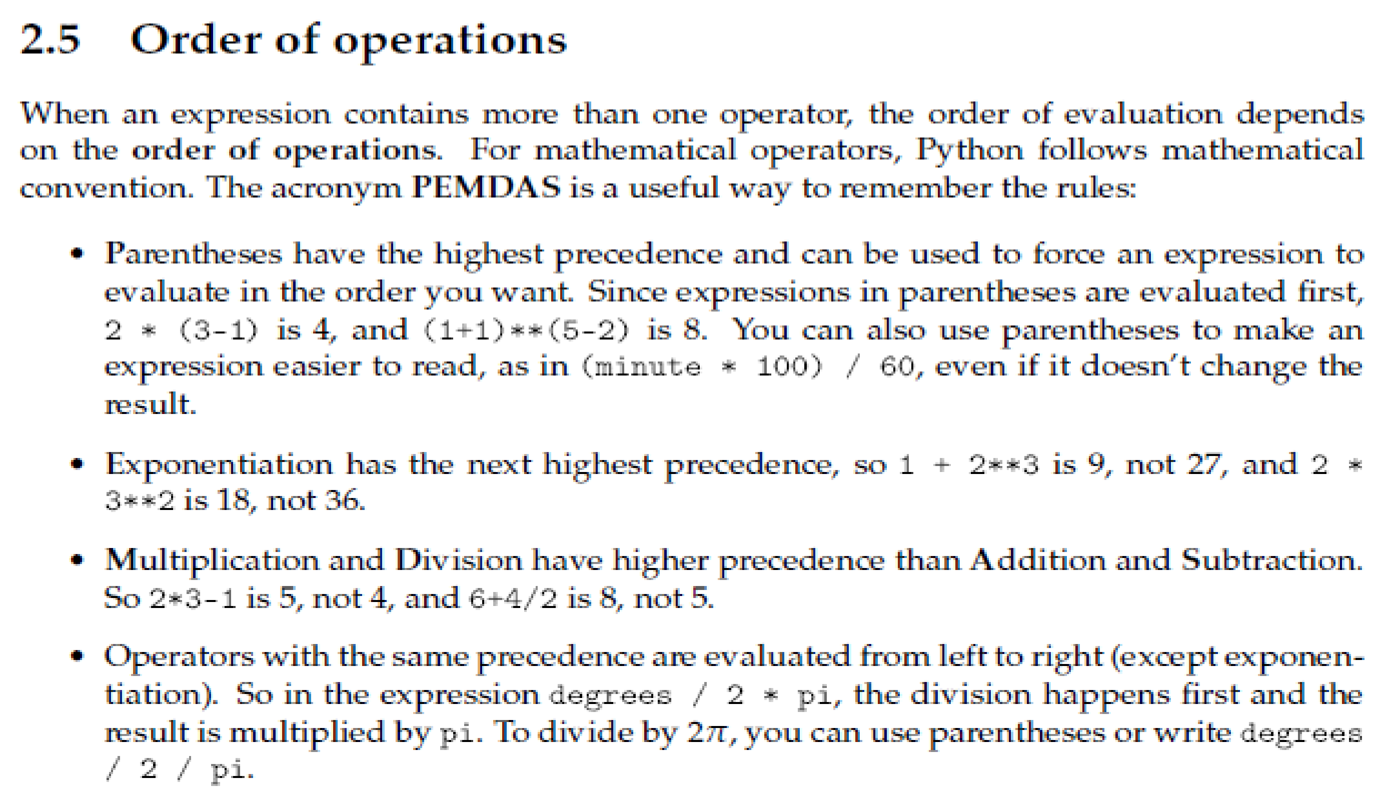
You can delete a single object or multiple objects by using the del statement. For example:

```
del var

del var_a, var_b
```

Python supports four different numerical types:

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

VARIABLE TYPES



2.5 Order of operations

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

- Parentheses 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. You can also use parentheses to make an expression easier to read, as in `(minute * 100) / 60`, even if it doesn't change the result.
- Exponentiation has the next highest precedence, so `1 + 2**3` is 9, not 27, and `2 * 3**2` is 18, not 36.
- Multiplication and Division have higher precedence than Addition and Subtraction. So `2*3-1` is 5, not 4, and `6+4/2` is 8, not 5.
- Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression `degrees / 2 * pi`, the division happens first and the result is multiplied by `pi`. To divide by 2π, you can use parentheses or write `degrees / 2 / pi`.

VARIABLE TYPES

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example:

```
#!/usr/bin/python


str = 'Hello World!'
print str           # Prints complete string
print str[0]        # Prints first character of the string
print str[2:5]      # Prints characters starting from 3rd to 5th
print str[2:]       # Prints string starting from 3rd character
print str * 2       # Prints string two times
print str + "TEST" # Prints concatenated string
```

VARIABLE TYPES

Strings

To create string literals, enclose them in single, double, or triple quotes as follows:

a = “Hello World”
b = ‘Python is groovy’
c = “””What is footnote 5?”””

The same type of quote used to start a string must be used to terminate it.Triplequoted strings capture all the text that appears prior to the terminating triple quote, as opposed to single- and double-quoted strings, which must be specified on one logical line.Triple-quoted strings are useful when the contents of a string literal span multiple lines of text such as the following:

print ‘’’Content-type: text/html
<h1> Hello World </h1>
Click <a href=”http://www.python.org”>here</a>.
‘’’

Strings are sequences of characters indexed by integers, starting at zero.To extract a single character, use the indexing operator s[i] like this:

```
a = “Hello World”
b = a[4] # b = ‘o’
```

VARIABLE TYPES

Strings

To extract a substring, use the slicing operator s[i:j].This extracts all elements from s whose index k is in the range i <= k < j. If either index is omitted, the beginning or end of the string is assumed, respectively:

```
c = a[:5] # c = “Hello”
d = a[6:] # d = “World”
e = a[3:8] # e = “lo Wo”
```

Strings are concatenated with the plus (+) operator:

```
g = a + “ This is a test”
```

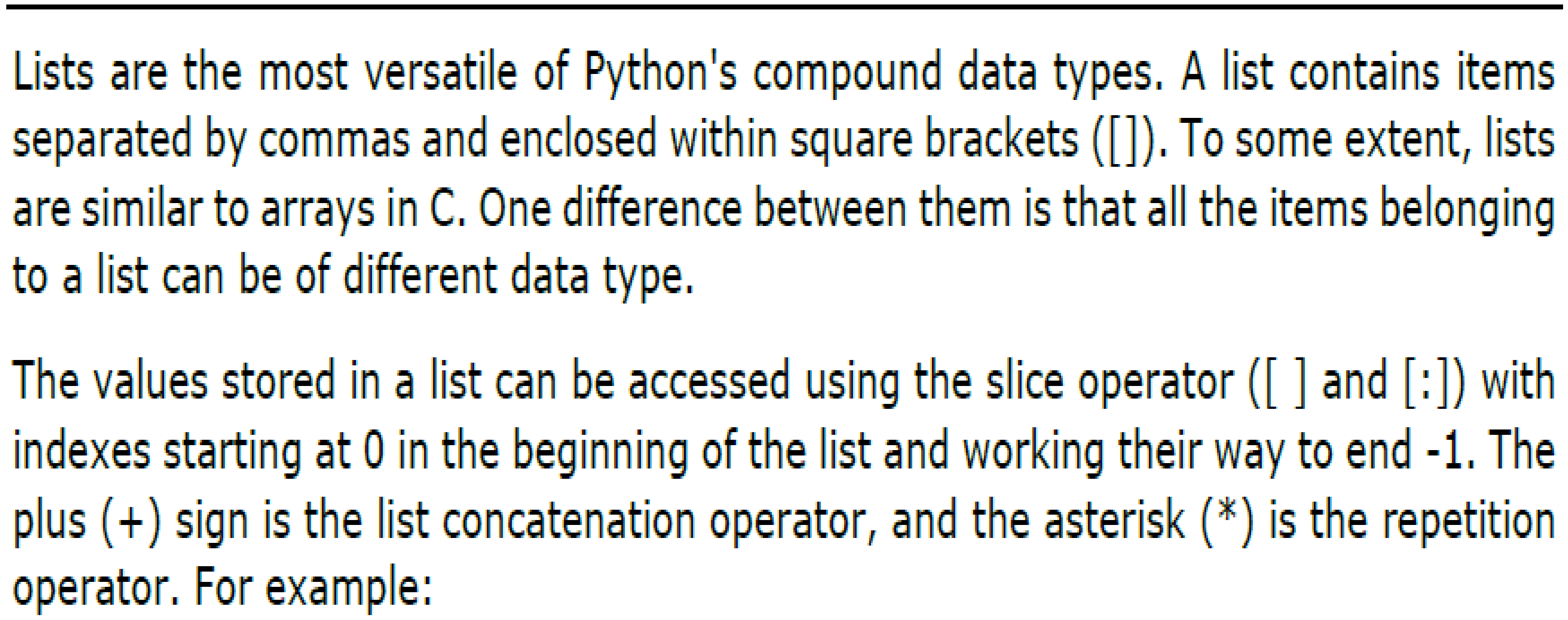
Other data types can be converted into a string by using either the str() or repr() function or backquotes (`), which are a shortcut notation for repr(). For example:

```
s = “The value of x is “ + str(x)
s = “The value of y is “ + repr(y)
s = “The value of y is “ + `y`
```

In many cases, str() and repr() return identical results. However, there are subtle differences
in semantics that are described in later chapters.

VARIABLE TYPES

Python Lists



Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example:

VARIABLE TYPES

Python Lists

Lists are sequences of arbitrary objects.You create a list as follows:

```
names = [ “Dave”, “Mark”, “Ann”, “Phil” ]
```

Lists are indexed by integers, starting with zero. Use the indexing operator to access and modify individual items of the list:

```
a = names[2] # Returns the third item of the list, “Ann”
names[0] = “Jeff” # Changes the first item to “Jeff”
```

To append new items to the end of a list, use the append() method:

```
names.append(“Kate”)
```

To insert an item in the list, use the insert() method:

```
names.insert(2, “Sydney”)
```

You can extract or reassign a portion of a list by using the slicing operator:

```
b = names[0:2] # Returns [ “Jeff”, “Mark” ]
c = names[2:] # Returns [ “Sydney”, “Ann”, “Phil”, “Kate” ]
names[1] = ‘Jeff’ # Replace the 2nd item in names with ‘Jeff’
names[0:2] = [‘Dave’,’Mark’,’Jeff’] # Replace the first two items of
# the list with the list on the right.
```

Use the plus (+) operator to concatenate lists:

```
a = [1,2,3] + [4,5] # Result is [1,2,3,4,5]
```

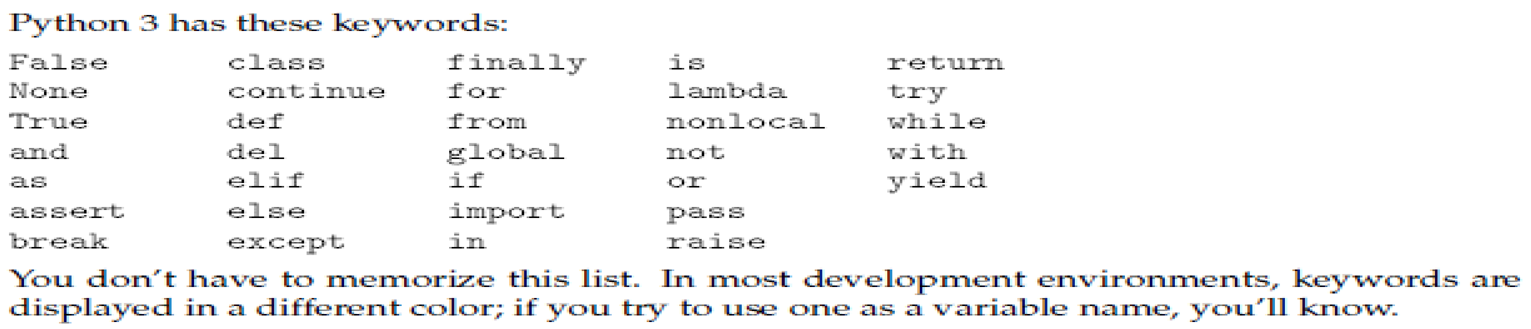
Lists can contain any kind of Python object, including other lists, as in the following example:

```
a = [1,”Dave”,3.14, [“Mark”, 7, 9, [100,101]], 10]
```

Nested lists are accessed as follows:

```
a[1] # Returns “Dave”
a[3][2] # Returns 9
a[3][3][1] # Returns 101
```

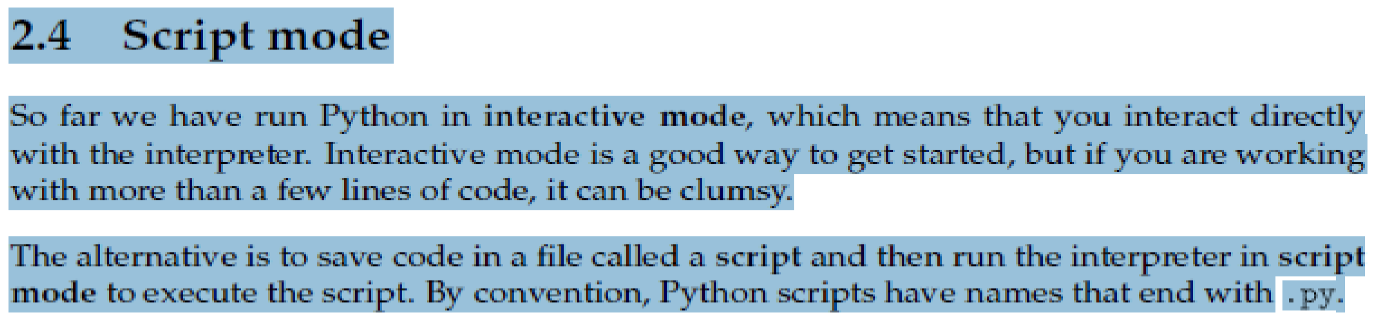
VARIABLE TYPES



Python 3 has these keywords:

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

You don't have to memorize this list. In most development environments, keywords are displayed in a different color; if you try to use one as a variable name, you'll know.



2.4 Script mode

So far we have run Python in **interactive mode**, which means that you interact directly with the interpreter. Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy.

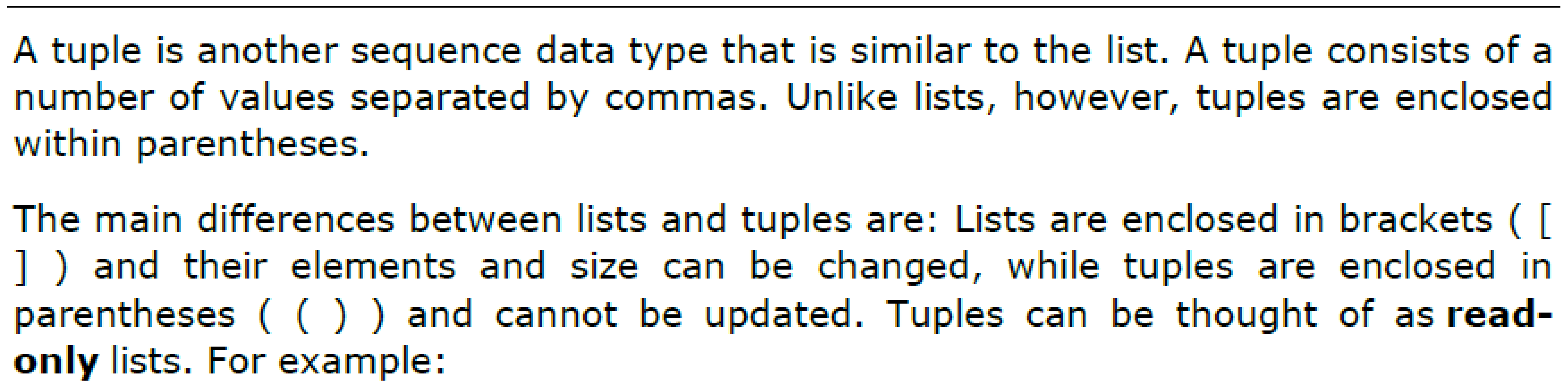
The alternative is to save code in a file called a script and then run the interpreter in **script mode** to execute the script. By convention, Python scripts have names that end with `.py`.

VARIABLE TYPES

```
#!/usr/bin/python


list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']


print list          # Prints complete list

print list[0]       # Prints first element of the list

print list[1:3]     # Prints elements starting from 2nd till 3rd

print list[2:]      # Prints elements starting from 3rd element

print tinylist * 2  # Prints list two times

print list + tinylist # Prints concatenated lists
```

VARIABLE TYPES

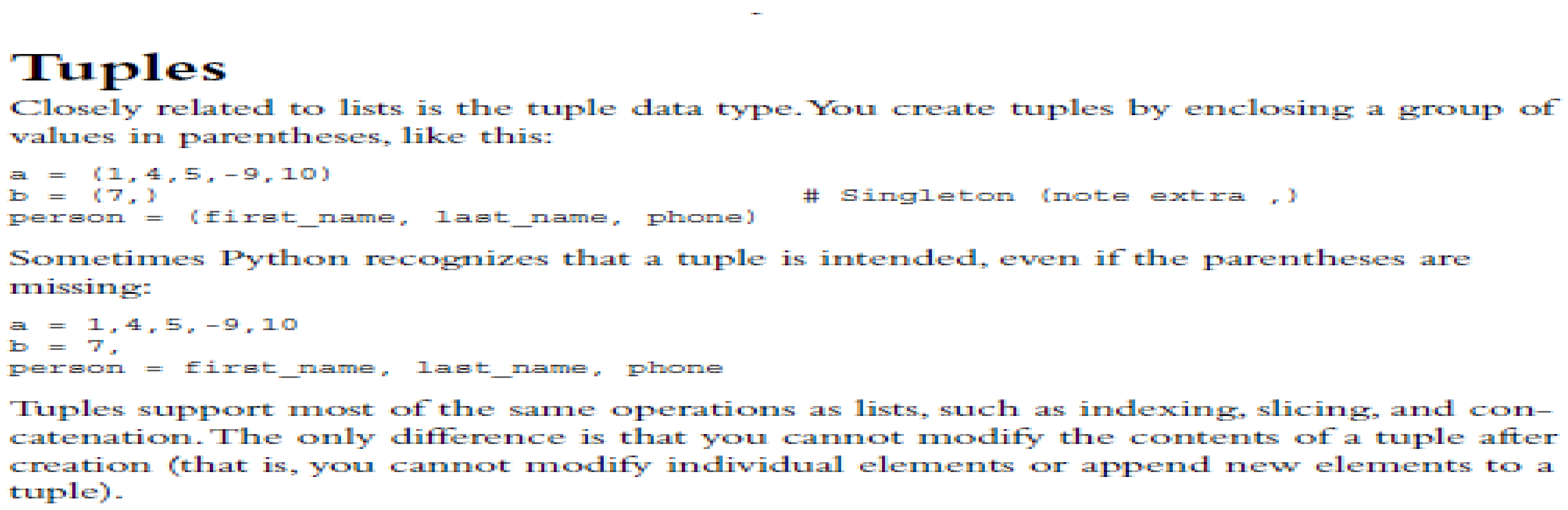
Python Tuples



A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as **read-only** lists. For example:

VARIABLE TYPES



Tuples

Closely related to lists is the tuple data type. You create tuples by enclosing a group of values in parentheses, like this:

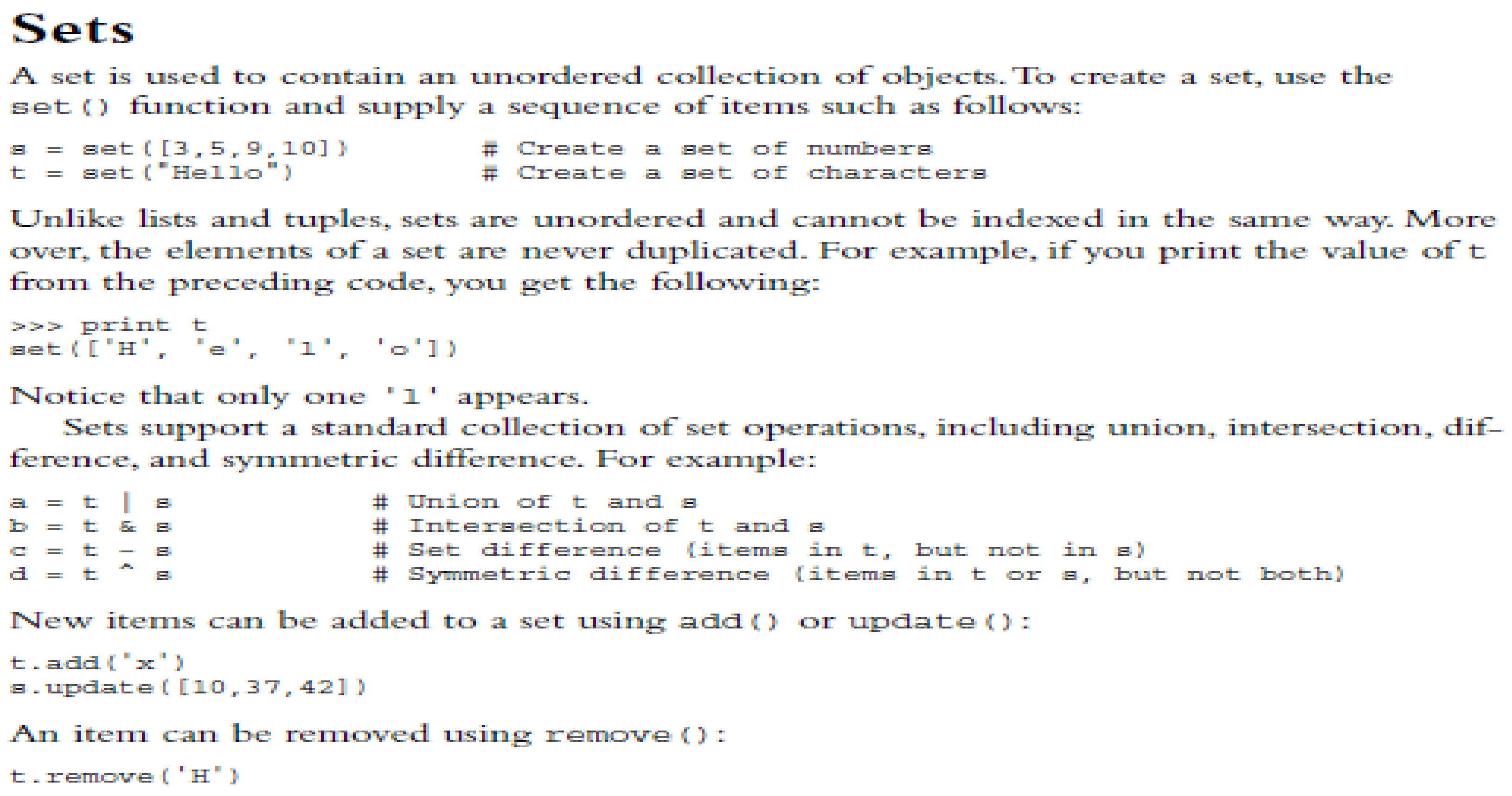
```
a = (1,4,5,-9,10)
b = (7,)                                 # Singleton (note extra ,)
person = (first_name, last_name, phone)
```

Sometimes Python recognizes that a tuple is intended, even if the parentheses are missing:

```
a = 1,4,5,-9,10
b = 7,
person = first_name, last_name, phone
```

Tuples support most of the same operations as lists, such as indexing, slicing, and concatenation. The only difference is that you cannot modify the contents of a tuple after creation (that is, you cannot modify individual elements or append new elements to a tuple).

VARIABLE TYPES



Sets

A set is used to contain an unordered collection of objects. To create a set, use the set() function and supply a sequence of items such as follows:

```
s = set([3,5,9,10])         # Create a set of numbers
t = set("Hello")            # Create a set of characters
```

Unlike lists and tuples, sets are unordered and cannot be indexed in the same way. More over, the elements of a set are never duplicated. For example, if you print the value of t from the preceding code, you get the following:

```
>>> print t
set(['H', 'e', 'l', 'o'])
```

Notice that only one 'l' appears.

Sets support a standard collection of set operations, including union, intersection, difference, and symmetric difference. For example:

```
a = t | s           # Union of t and s
b = t & s           # Intersection of t and s
c = t – s           # Set difference (items in t, but not in s)
d = t ^ s           # Symmetric difference (items in t or s, but not both)
```

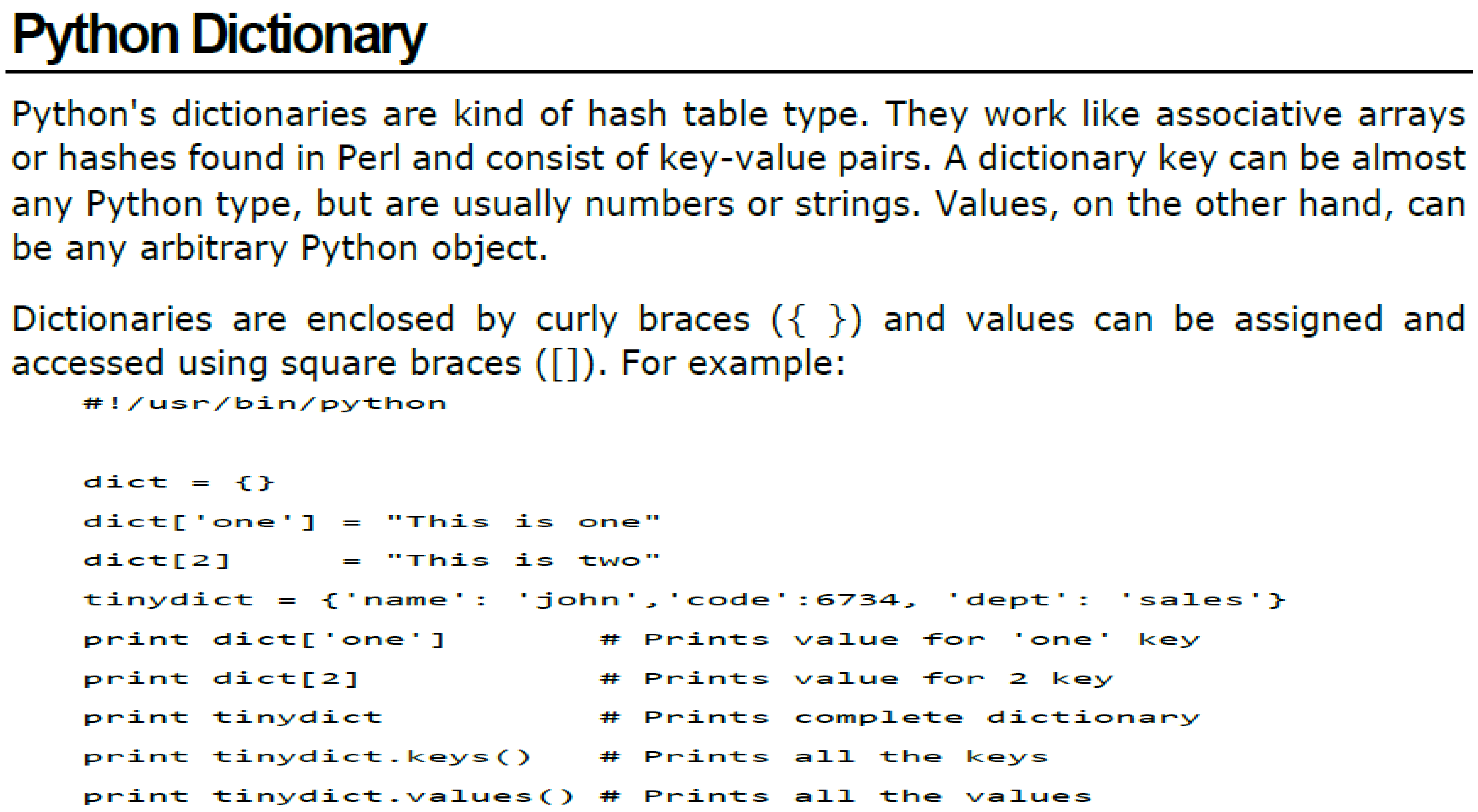
New items can be added to a set using add() or update():

```
t.add('x')
s.update([10,37,42])
```

An item can be removed using remove():

```
t.remove('H')
```

VARIABLE TYPES



Python Dictionary

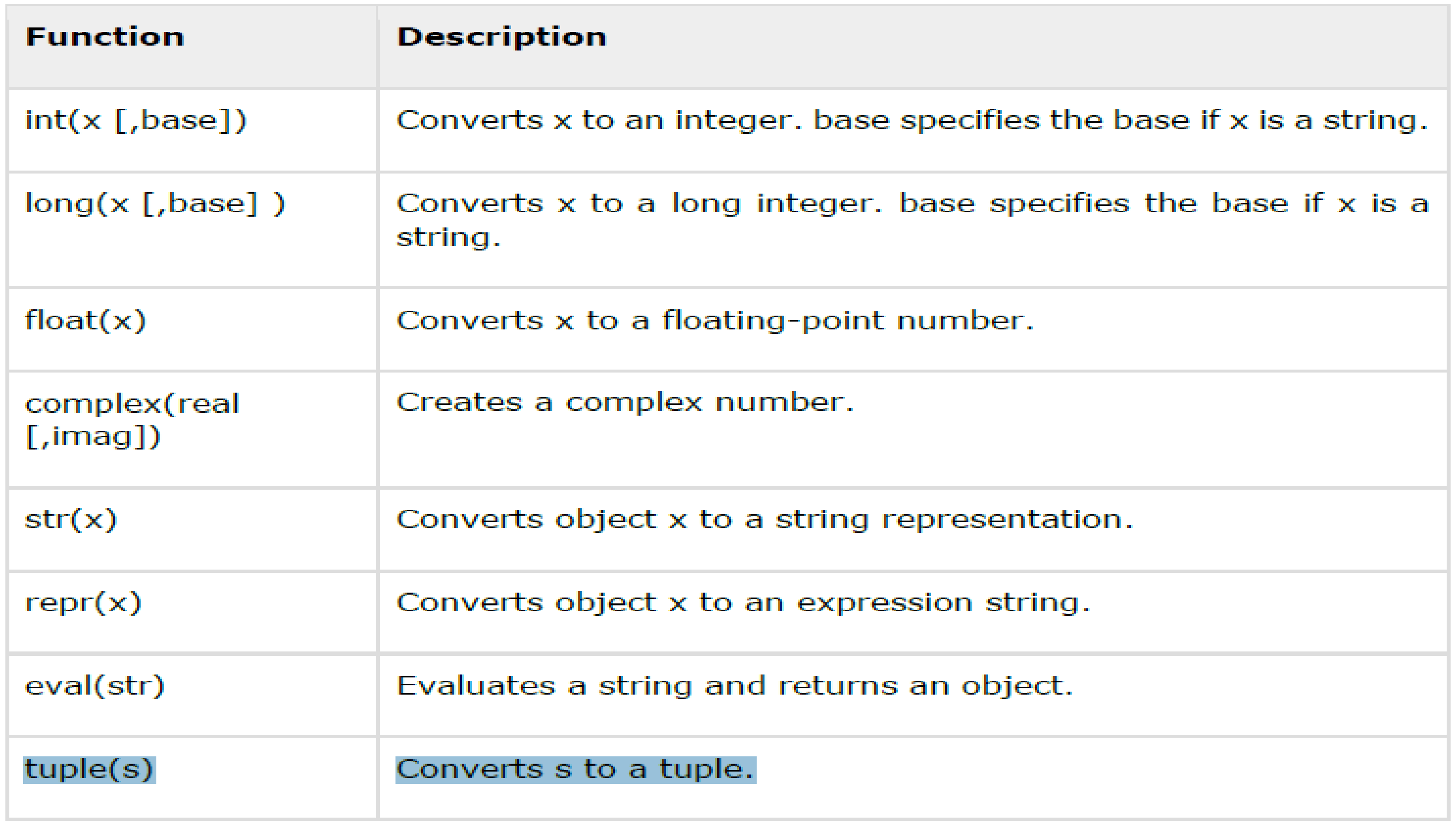
Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example:

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

VARIABLE TYPES

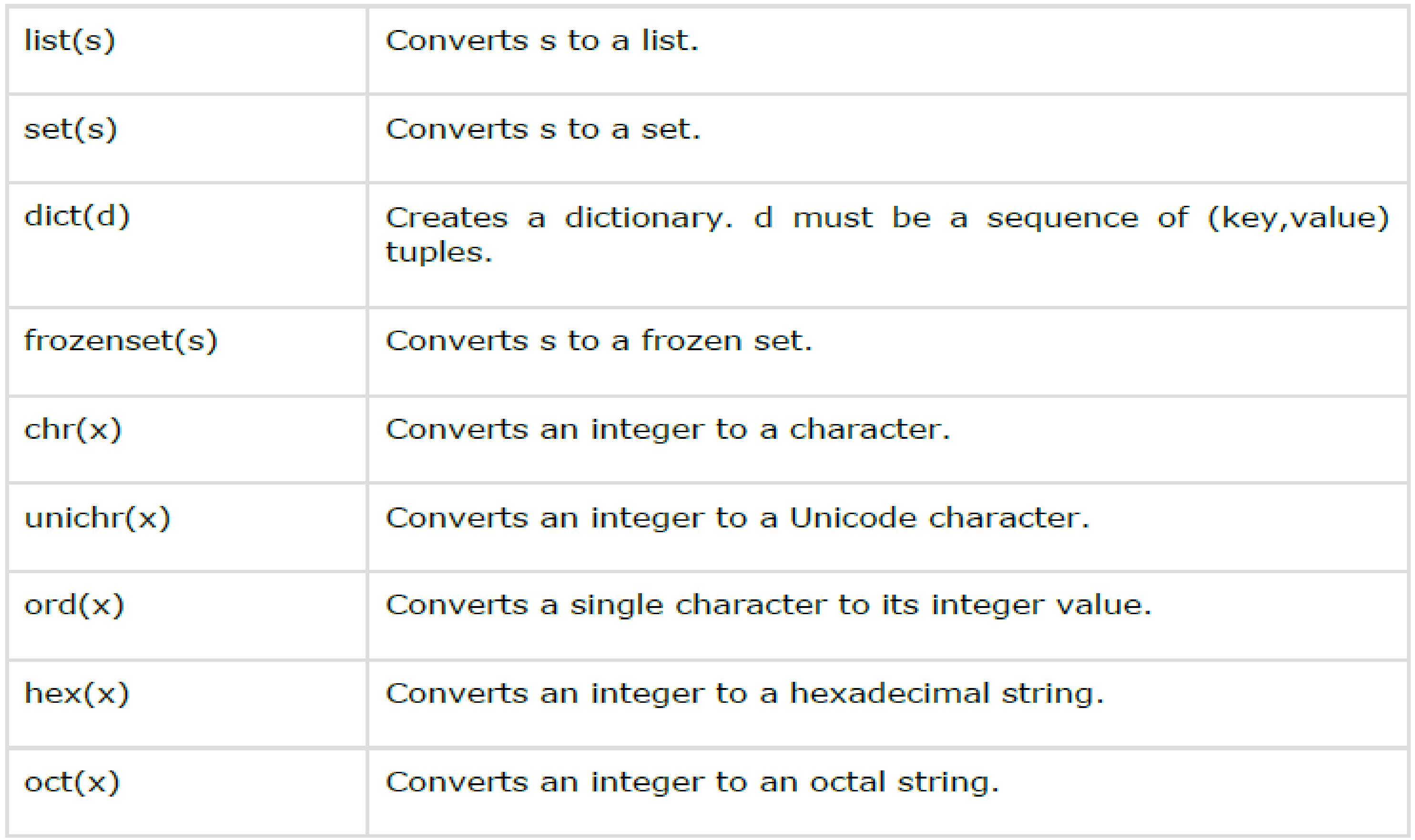
Data Type Conversion



Function	Description
int(x [,base])	Converts x to an integer. base specifies the base if x is a string.
long(x [,base])	Converts x to a long integer. base specifies the base if x is a string.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.

VARIABLE TYPES

Data Type Conversion



list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.

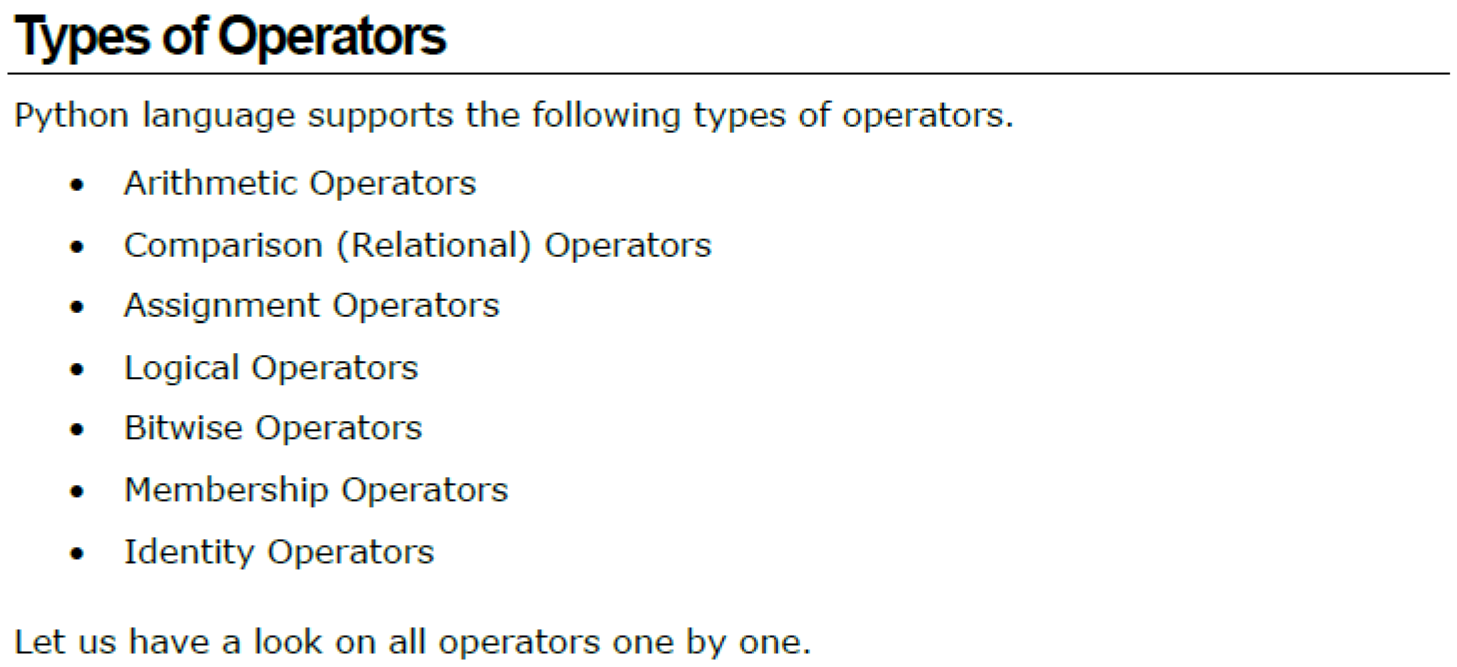
VARIABLE TYPES

Frozensets are like sets except that they cannot be changed, i.e. they are immutable:

sets :- sets are mutable:

str() is used for creating output for end user while repr() is mainly used for debugging and development. repr's goal is to be unambiguous and str's is to be readable

Operators and Expressions



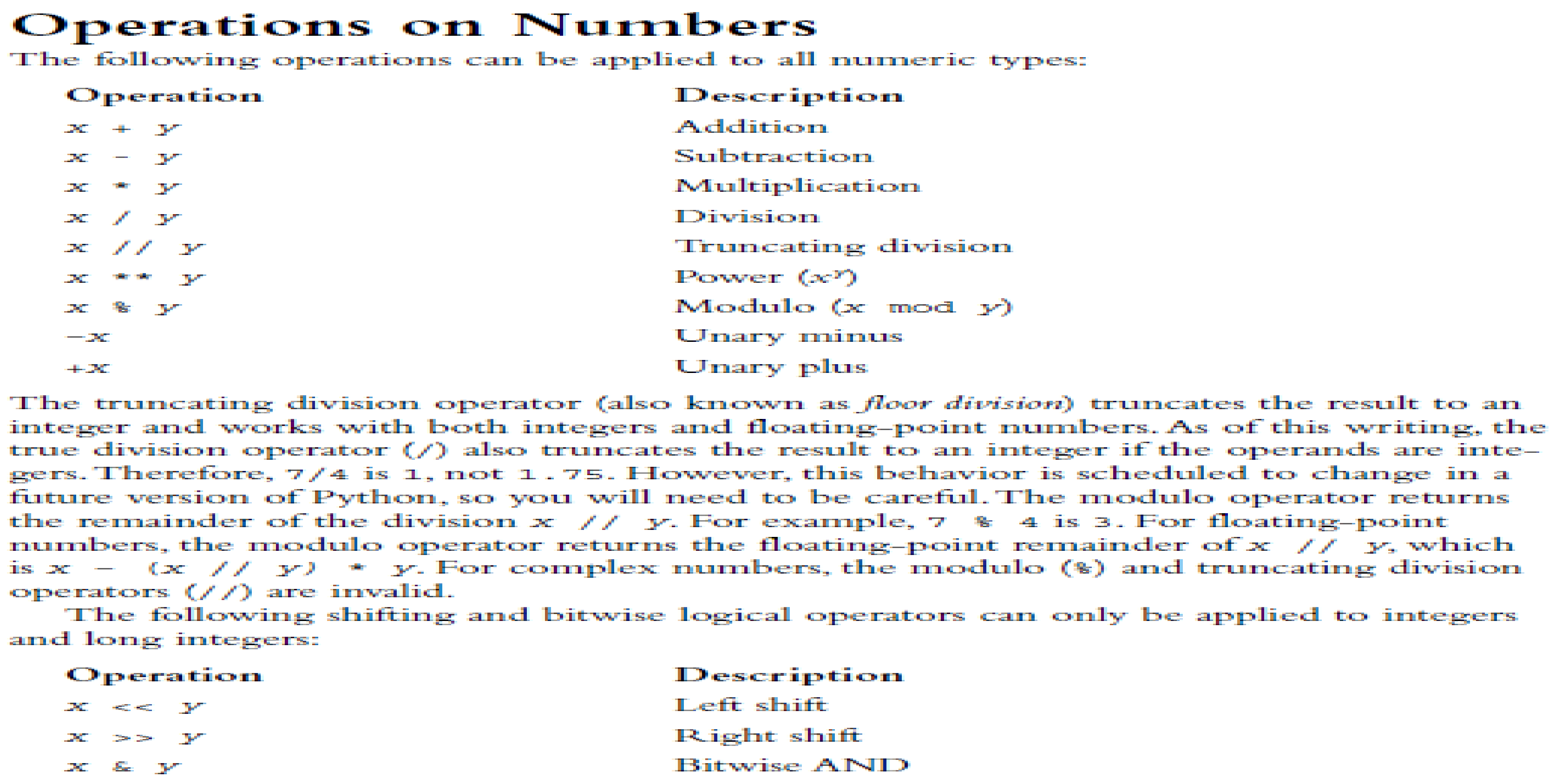
Types of Operators

Python language supports the following types of operators.

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

Let us have a look on all operators one by one.

Operators and Expressions



Operations on Numbers

The following operations can be applied to all numeric types:

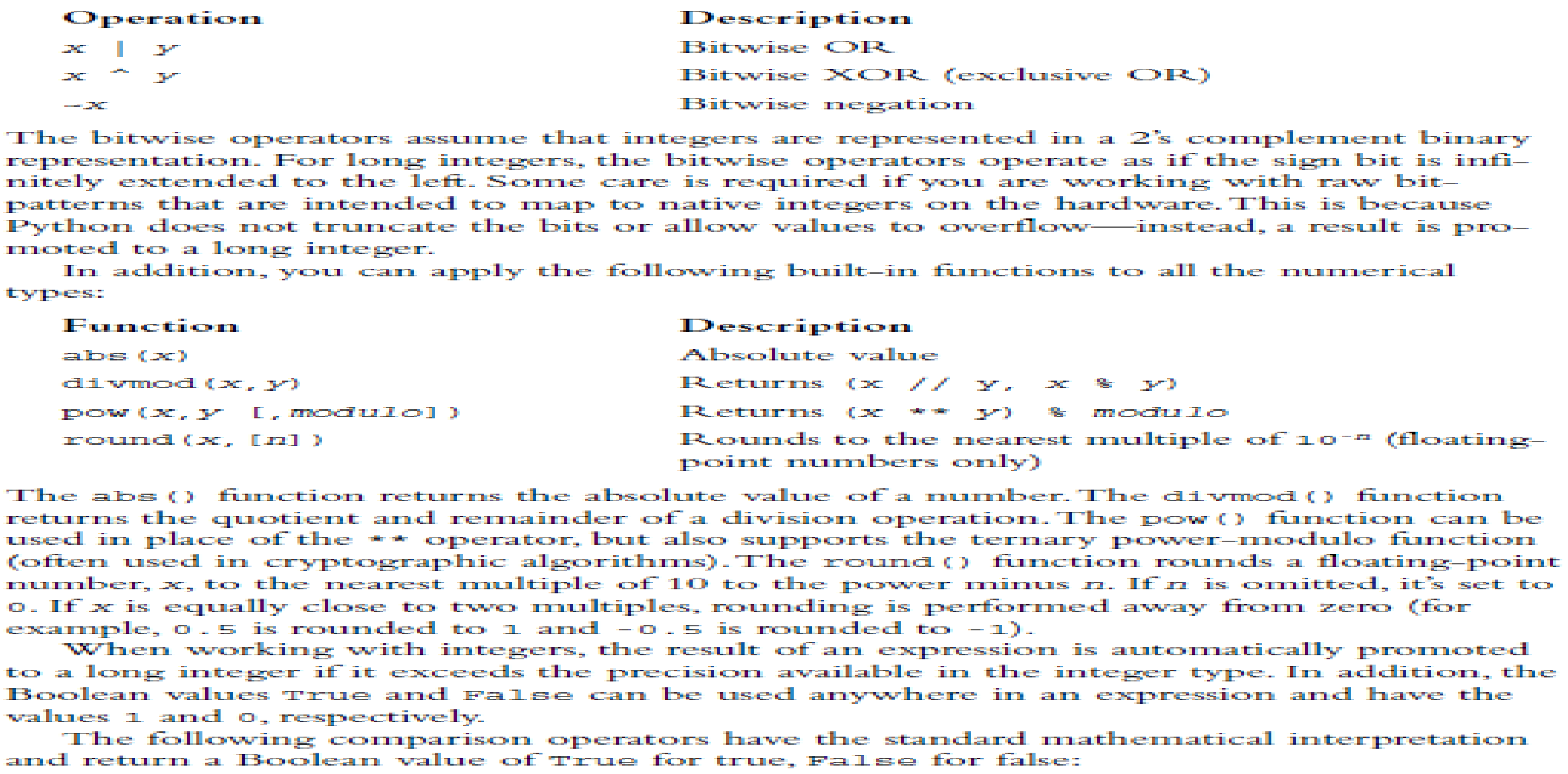
Operation	Description
x + *y*	Addition
x - *y*	Subtraction
x * *y*	Multiplication
x / *y*	Division
x // *y*	Truncating division
x ** *y*	Power (x^y)
x % *y*	Modulo (*x* mod *y*)
–*x*	Unary minus
+*x*	Unary plus

The truncating division operator (also known as *floor division*) truncates the result to an integer and works with both integers and floating-point numbers. As of this writing, the true division operator (/) also truncates the result to an integer if the operands are integers. Therefore, 7/4 is 1, not 1.75. However, this behavior is scheduled to change in a future version of Python, so you will need to be careful. The modulo operator returns the remainder of the division *x* // *y*. For example, 7 % 4 is 3. For floating-point numbers, the modulo operator returns the floating-point remainder of *x* // *y*, which is *x* – (*x* // *y*) * *y*. For complex numbers, the modulo (%) and truncating division operators (//) are invalid.

The following shifting and bitwise logical operators can only be applied to integers and long integers:

Operation	Description
x << *y*	Left shift
x >> *y*	Right shift
x & *y*	Bitwise AND

Operators and Expressions



Operation	Description
`x	y`	Bitwise OR
`x ^ y`	Bitwise XOR (exclusive OR)
`~x`	Bitwise negation

The bitwise operators assume that integers are represented in a 2's complement binary representation. For long integers, the bitwise operators operate as if the sign bit is infinitely extended to the left. Some care is required if you are working with raw bit-patterns that are intended to map to native integers on the hardware. This is because Python does not truncate the bits or allow values to overflow—instead, a result is promoted to a long integer.

In addition, you can apply the following built-in functions to all the numerical types:

Function	Description
`abs(x)`	Absolute value
`divmod(x,y)`	Returns `(x // y, x % y)`
`pow(x,y [,modulo])`	Returns `(x ** y) % modulo`
`round(x, [n])`	Rounds to the nearest multiple of 10^{-n} (floating-point numbers only)

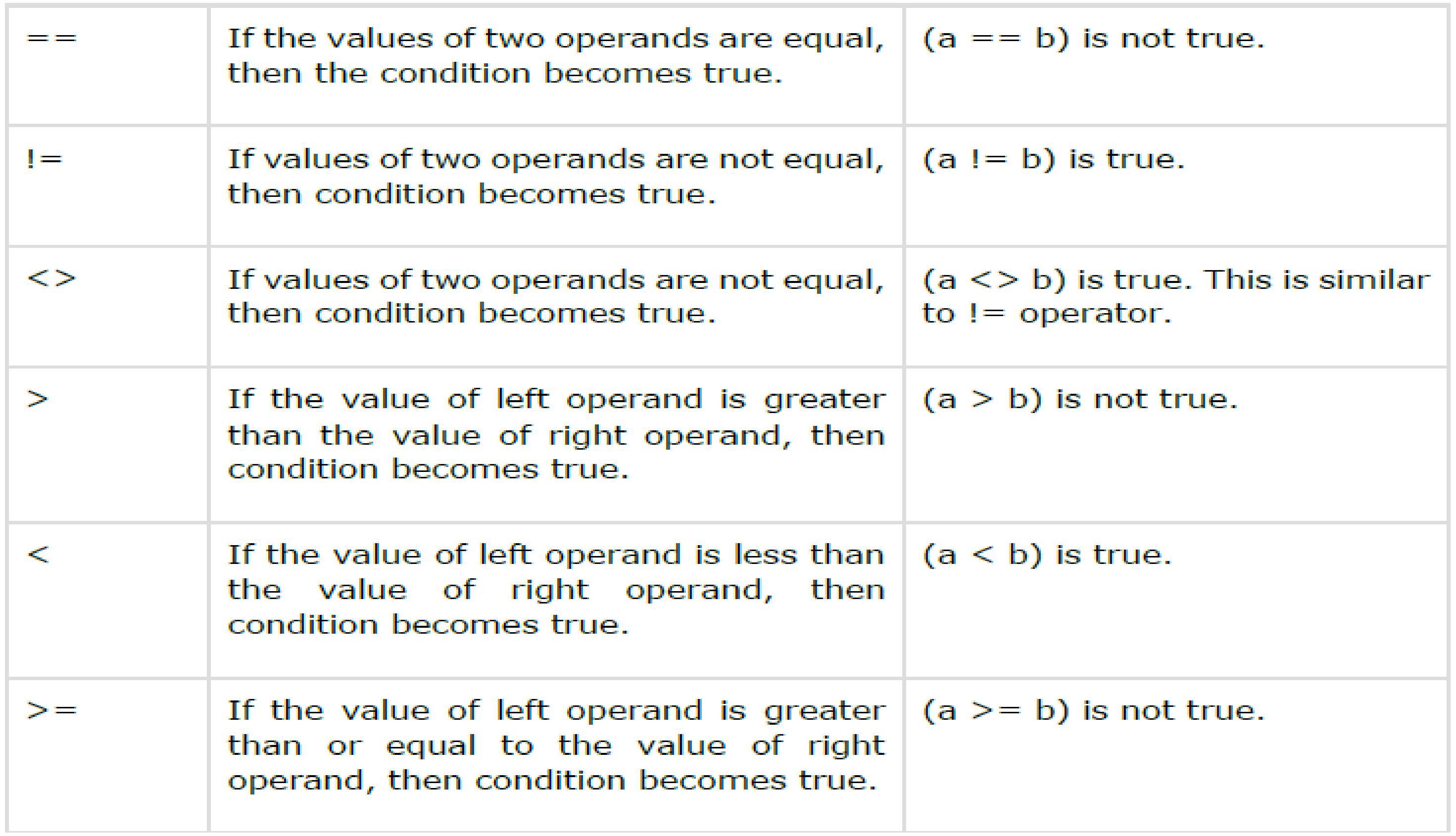
The `abs()` function returns the absolute value of a number. The `divmod()` function returns the quotient and remainder of a division operation. The `pow()` function can be used in place of the `**` operator, but also supports the ternary power-modulo function (often used in cryptographic algorithms). The `round()` function rounds a floating-point number, *x*, to the nearest multiple of 10 to the power minus *n*. If *n* is omitted, it's set to `0`. If *x* is equally close to two multiples, rounding is performed away from zero (for example, `0.5` is rounded to `1` and `-0.5` is rounded to `-1`).

When working with integers, the result of an expression is automatically promoted to a long integer if it exceeds the precision available in the integer type. In addition, the Boolean values `True` and `False` can be used anywhere in an expression and have the values `1` and `0`, respectively.

The following comparison operators have the standard mathematical interpretation and return a Boolean value of `True` for true, `False` for false:

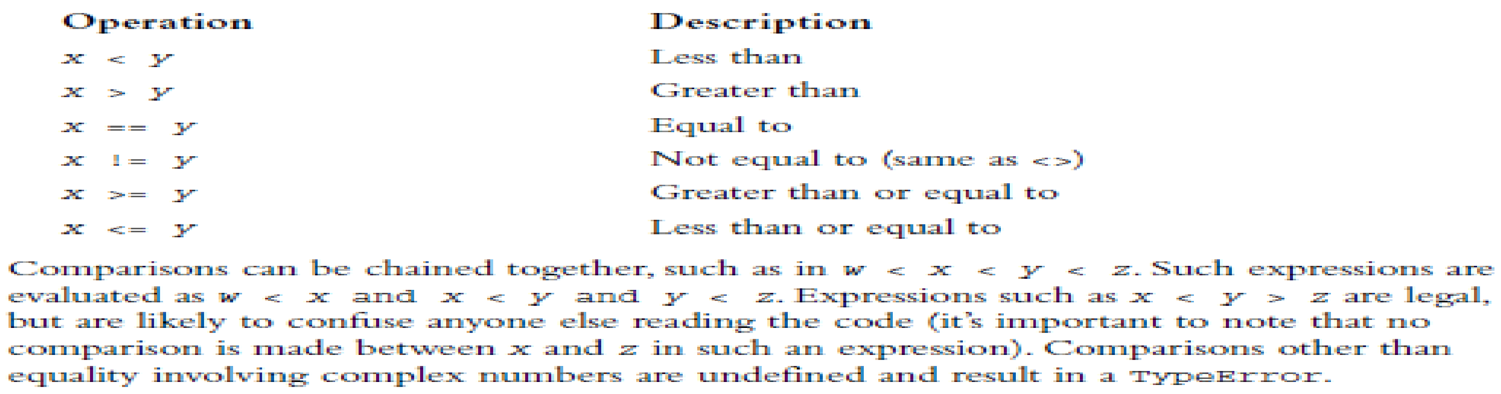
Operators and Expressions

Python Comparison Operators



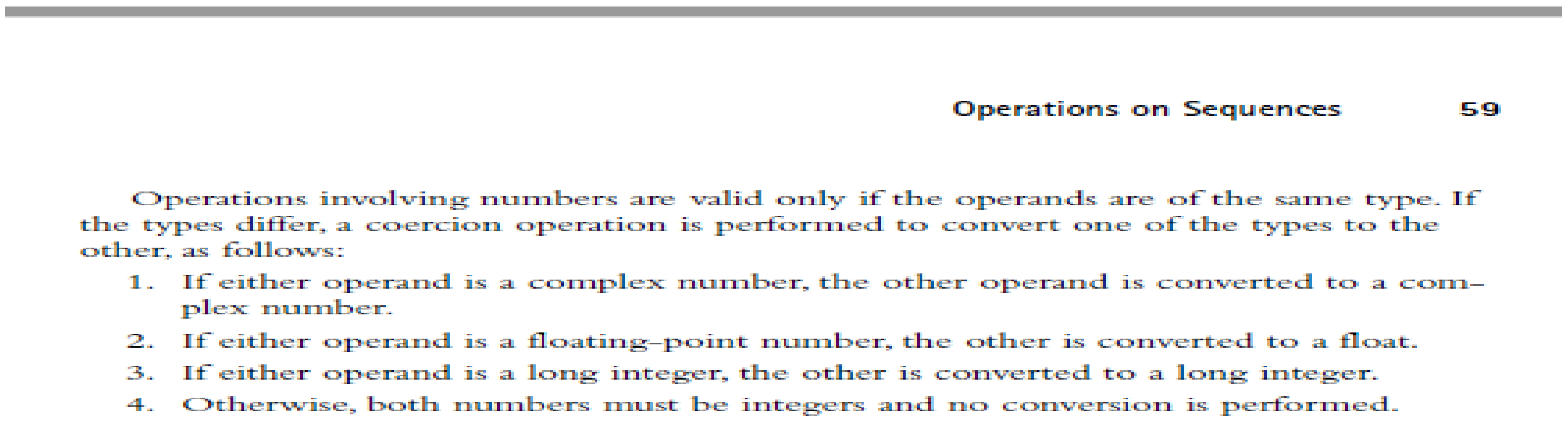
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.

Operators and Expressions



Operation	Description
`x < y`	Less than
`x > y`	Greater than
`x == y`	Equal to
`x != y`	Not equal to (same as <>)
`x >= y`	Greater than or equal to
`x <= y`	Less than or equal to

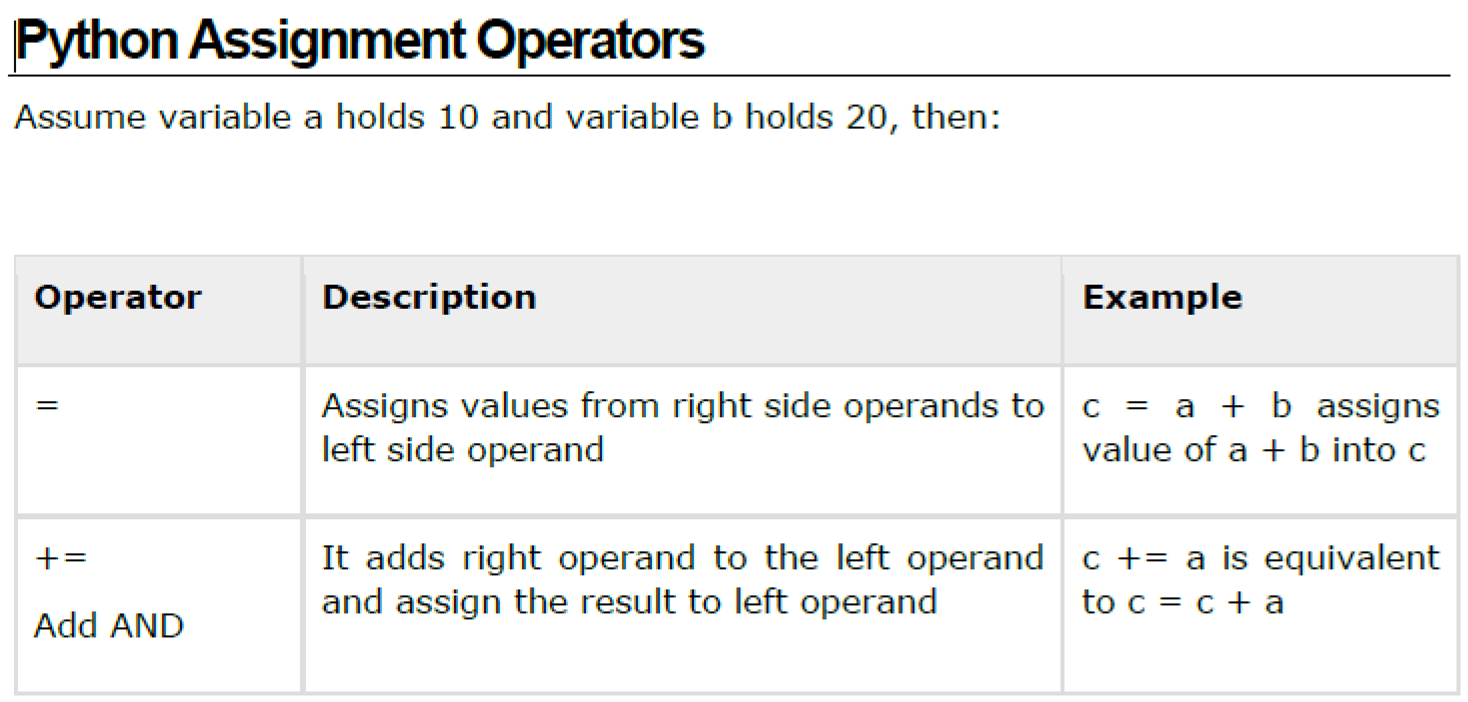
Comparisons can be chained together, such as in `w < x < y < z`. Such expressions are evaluated as `w < x and x < y and y < z`. Expressions such as `x < y > z` are legal, but are likely to confuse anyone else reading the code (it's important to note that no comparison is made between `x` and `z` in such an expression). Comparisons other than equality involving complex numbers are undefined and result in a `TypeError`.



Operations involving numbers are valid only if the operands are of the same type. If the types differ, a coercion operation is performed to convert one of the types to the other, as follows:

1. If either operand is a complex number, the other operand is converted to a complex number.
2. If either operand is a floating-point number, the other is converted to a float.
3. If either operand is a long integer, the other is converted to a long integer.
4. Otherwise, both numbers must be integers and no conversion is performed.

Operators and Expressions

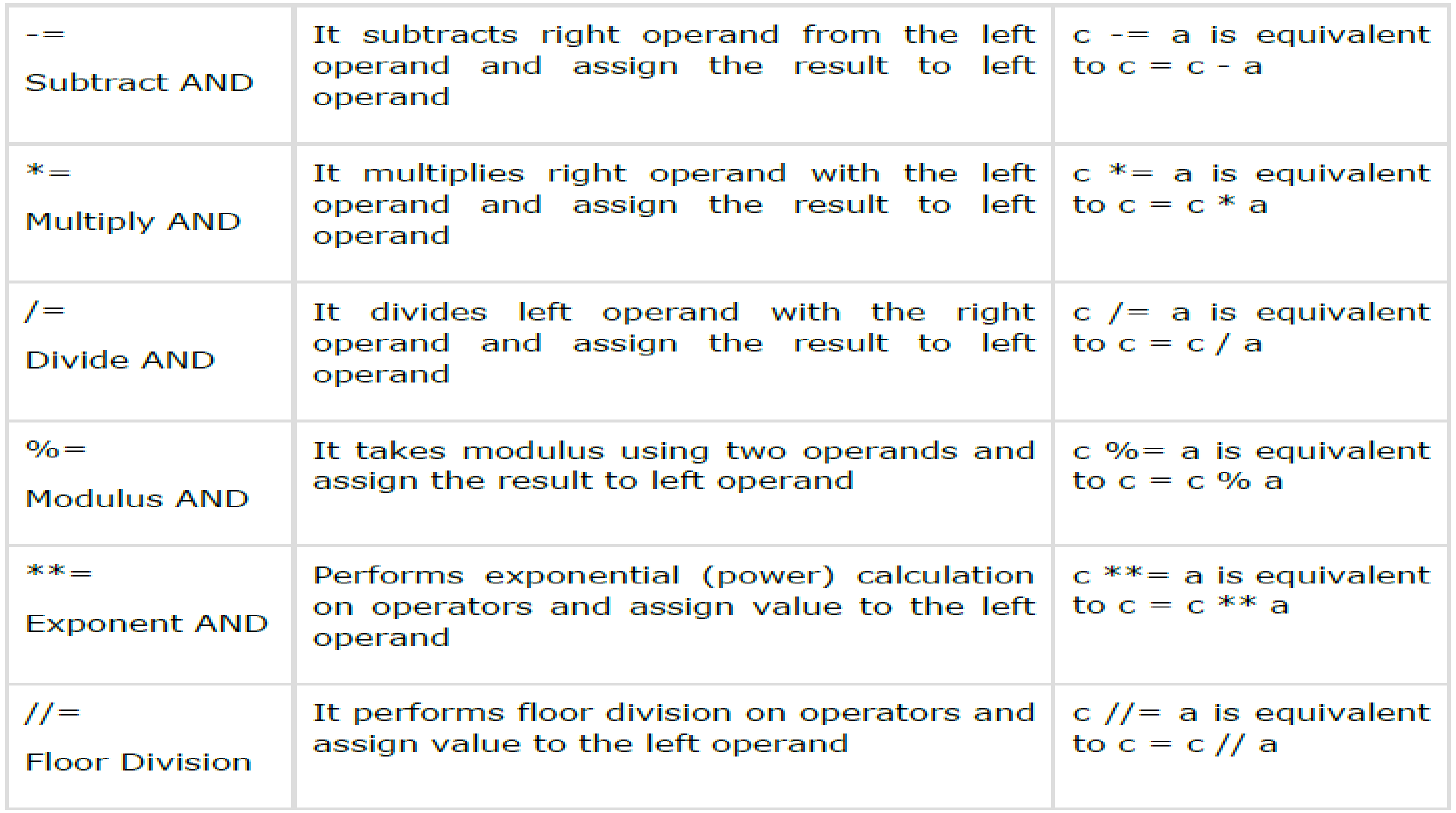


Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then:

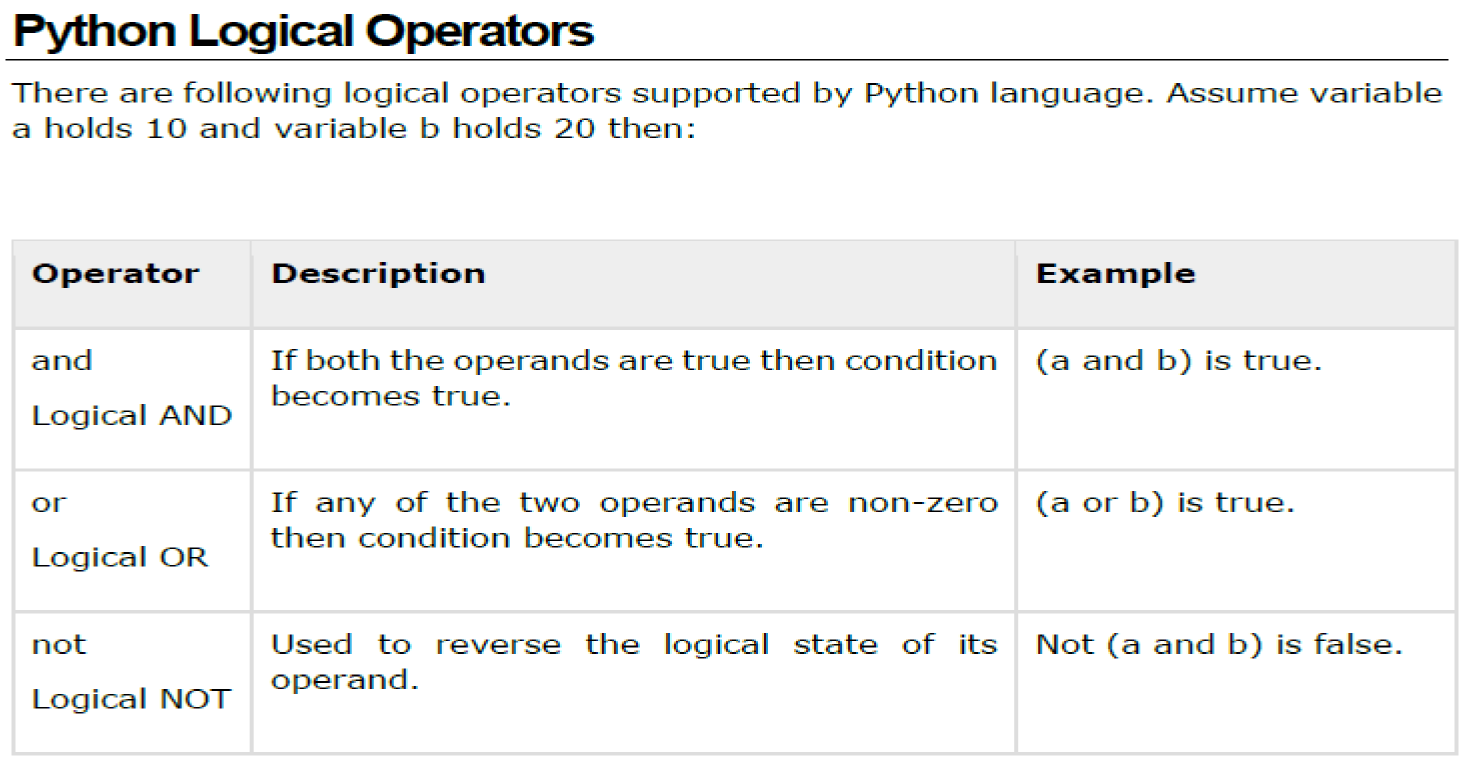
Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a

Operators and Expressions



-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Operators and Expressions



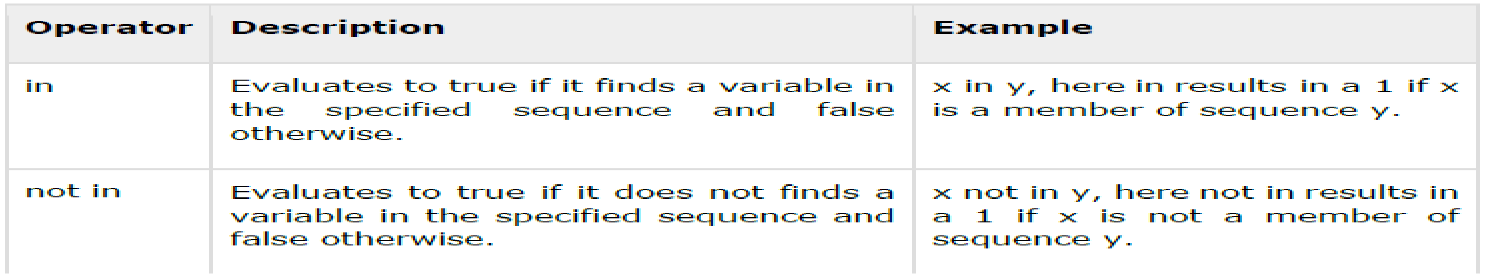
Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then:

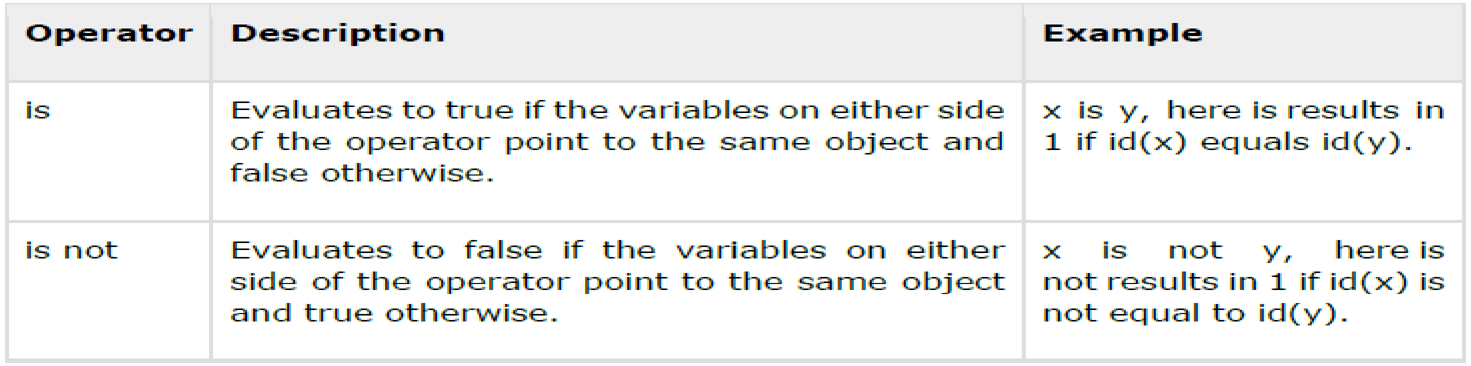
Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not (a and b) is false.

Operators and Expressions

Python Membership Operators

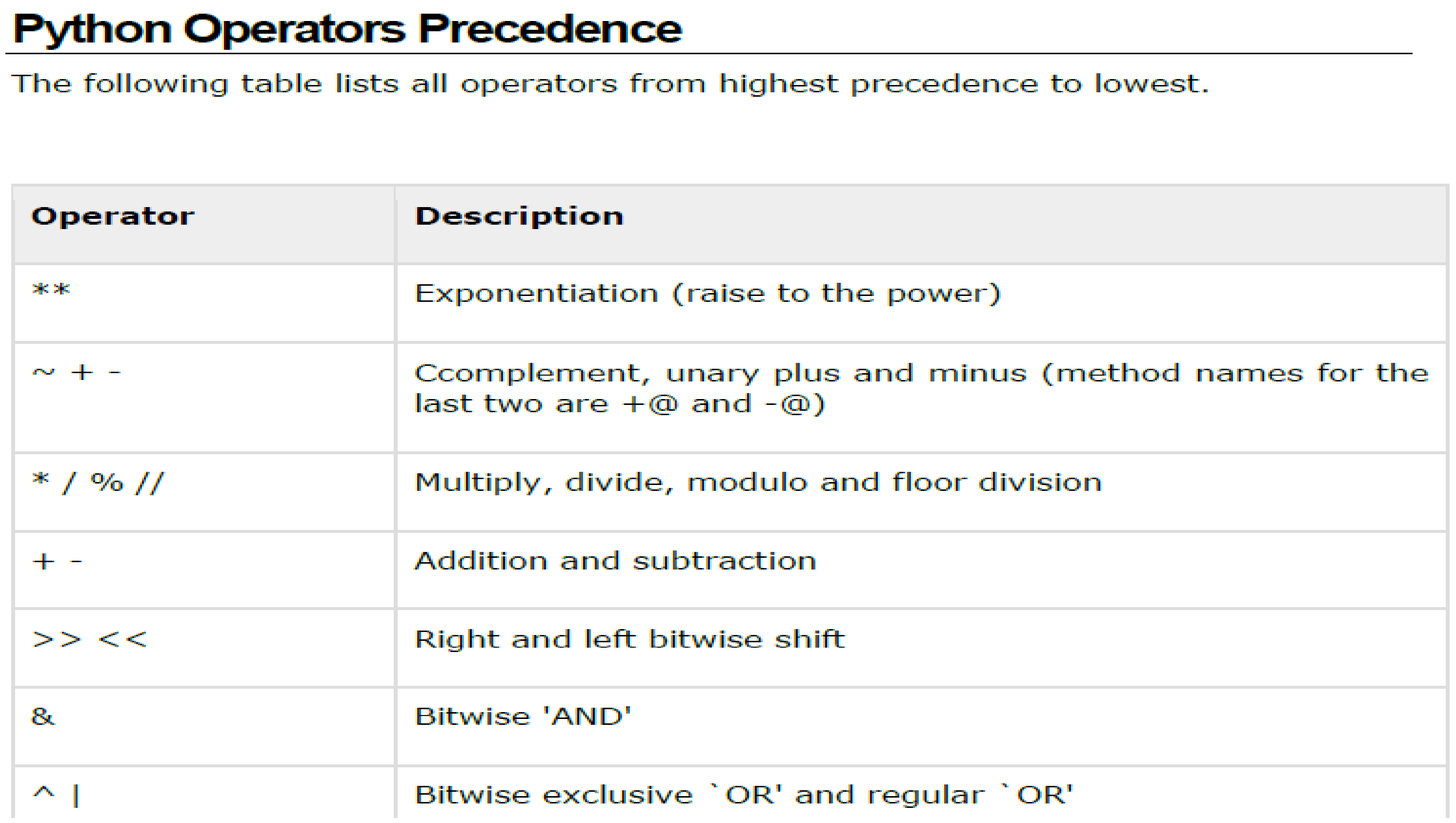


Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.



Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Operators and Expressions

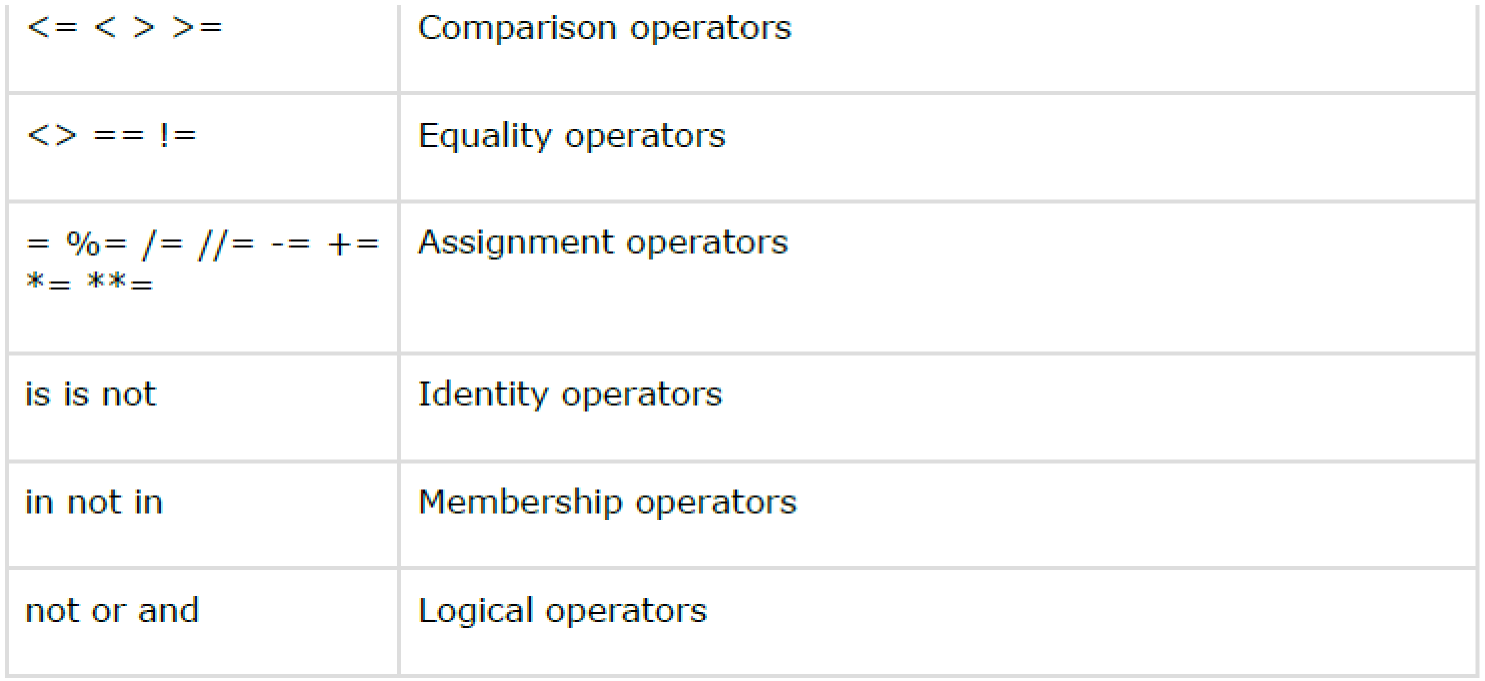


Python Operators Precedence

The following table lists all operators from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'

Operators and Expressions



<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Thank you