# Python Language Basics

## FOLLOW PEP8!

In [1]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

In [16]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import statsmodels as sm
```

## Language Semantics

Indentation, not braces: Use **four spaces** as your default indentation and replacing tabs with four spaces.

In [17]:
```
"""for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)"""
```

Out[17]:
```
'for x in array:\n    if x < pivot:\n        less.append(x)\n    else:\n
greater.append(x)'
```

Python statements also do not need to be terminated by semicolons.

Semicolons can be used, however, to separate multiple statements on a single line (Putting multiple statements on one line is generally discouraged in Python):

In [120]:
```
a = 5; b = 6; c = 7
```

**Multi-Line Statements**

In [126]:
```
one = 2
two = 3
three = 5
```

In [127]:
```
total = one + \
        two + \
        three
```

In [128]:
```
total
```

Out[128]:
```
10
```

In [ ]:
```

```

Everything is an object

## Comments

Any text preceded by the hash mark (pound sign) **#** is ignored by the Python interpreter.

In [19]:
```
print("Reached this line") # Simple status report
```

```
Reached this line
```

## Variables and argument passing

In [20]:
```
a = [1, 2, 3]
```

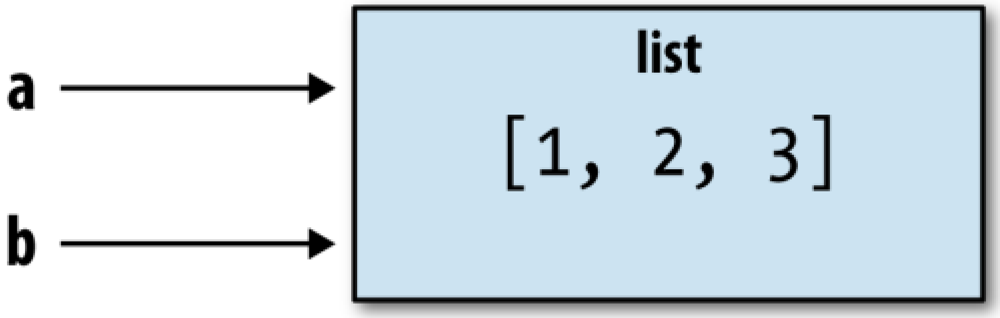
When assigning a variable (or name) in Python, you are creating a **reference** to the object on the righthand side of the equals sign.

In [21]:
```
b = a
```

```
In [22]: a.append(4)
```

```
In [23]: b
```

```
Out[23]: [1, 2, 3, 4]
```



When you pass objects as arguments to a function, new local variables are created ref- erencing the original objects without any copying. If you bind a new object to a vari- able inside a function, that change will not be reflected in the parent scope:

```
In [24]: def append_element(some_list, element):
             some_list.append(element)
```

```
In [25]: data = [1, 2, 3]
```

```
In [26]: append_element(data, 4)
```

```
In [27]: data
```

```
Out[27]: [1, 2, 3, 4]
```

## Dynamic references, strong types

Object references in Python have no type associated with them:

```
In [28]: a = 5
         type(a)
```

```
Out[28]: int
```

```
In [29]: a = 'foo'
         type(a)
```

```
Out[29]: str
```

Python is considered a strongly typed language:

```
In [30]: ## '5' + 5 error
```

Implicit conversions will occur only in certain obvious circumstances, such as the following:

```
In [31]: a = 4.5; b = 2
```

```
In [32]: print('a is {0}, b is {1}'.format(type(a), type(b)))
```

```
a is <class 'float'>, b is <class 'int'>
```

```
In [33]: a / b
```

```
Out[33]: 2.25
```

You can check that an object is an instance of a particular type using the **isinstance** function:

```
In [34]: isinstance(a, int)
```

```
Out[34]: False
```

**isinstance** can accept a tuple of types if you want to check that an object's type is among those present in the tuple:

```
In [35]: a = 5; b = 4.5
```

```
In [110]: isinstance(a, (int, float))
```

```
Out[110]: False
```

Assignments can be done on more than one variable "simultaneously" on the same line like this:

```
In [111]: a, b = 3, 4
```

```
In [ ]:
```

Swap can be done like this:

```
In [37]: a, b = 1, 2
```

```
In [38]: b, a = a, b
```

In [39]:
```
a, b
```
Out[39]:
```
(2, 1)
```

## Binary operators and comparisons

In [40]:
```
5 - 7
```
Out[40]:
```
-2
```

In [41]:
```
12 + 21.5
```
Out[41]:
```
33.5
```

In [42]:
```
5 <= 2
```
Out[42]:
```
False
```

To check if two references refer to the same object, use the **is** / **is not** keyword.

In [43]:
```
a = [1, 2, 3]
b = a
c = list(a)
```

In [44]:
```
a is b
```
Out[44]:
```
True
```

In [45]:
```
a is not c
```
Out[45]:
```
True
```

Comparing with is is not the same as the **==** operator:

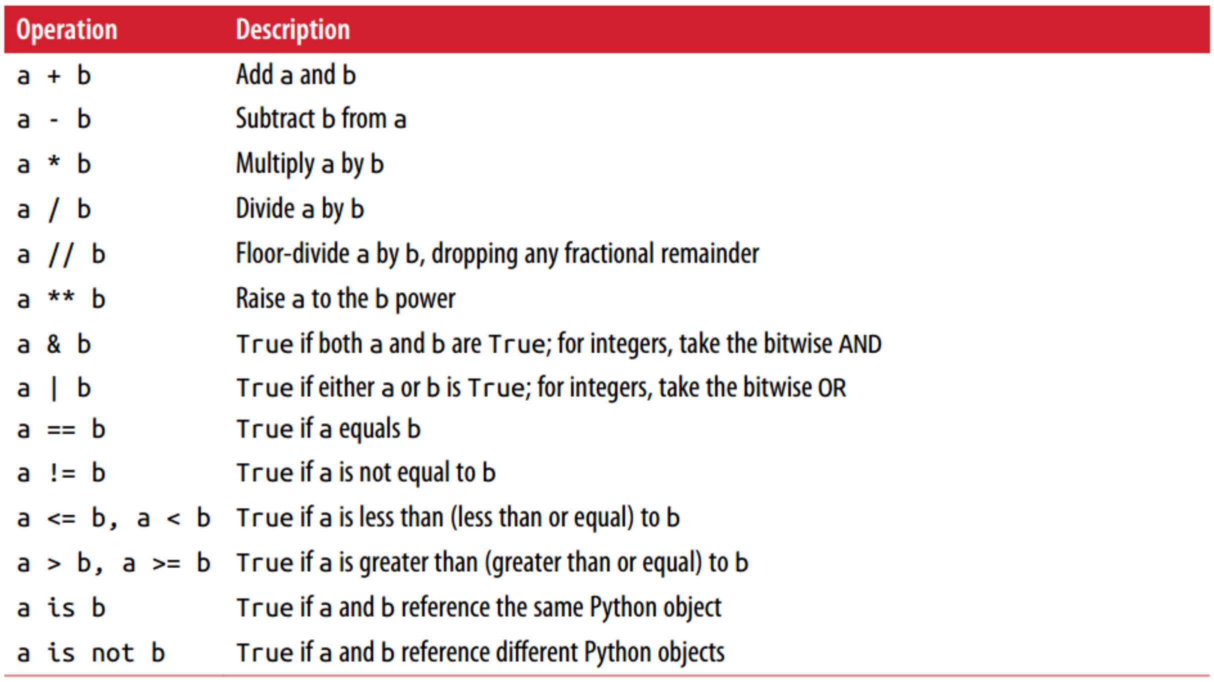
In [46]:
```
a == c
```
Out[46]:
```
True
```

A very common use of **is** and **is not** is to check if a variable is **None**:

In [47]:
```
a = None
```

In [48]:
```
a is None
```
Out[48]:
```
True
```

*Binary operators*

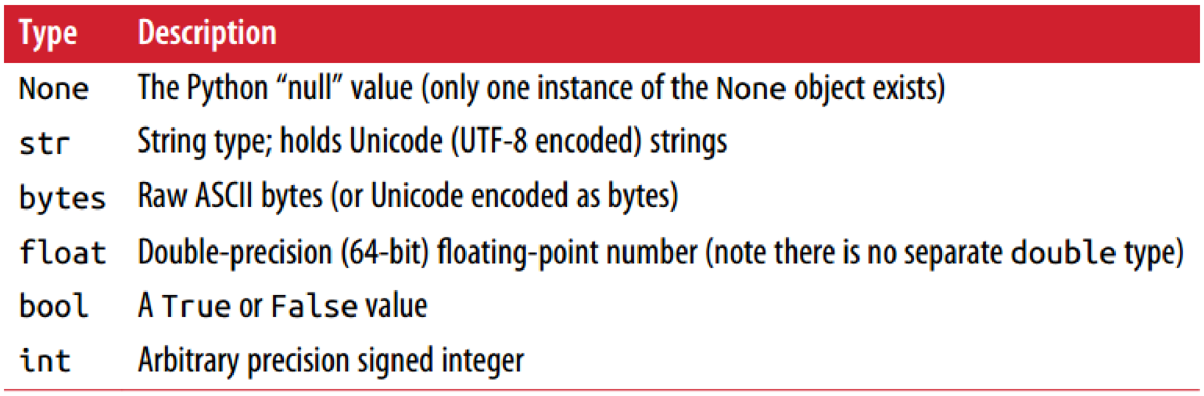


| Operation | Description |
|---|---|
| `a + b` | Add a and b |
| `a - b` | Subtract b from a |
| `a * b` | Multiply a by b |
| `a / b` | Divide a by b |
| `a // b` | Floor-divide a by b, dropping any fractional remainder |
| `a ** b` | Raise a to the b power |
| `a & b` | `True` if both a and b are `True`; for integers, take the bitwise AND |
| `a \| b` | `True` if either a or b is `True`; for integers, take the bitwise OR |
| `a == b` | `True` if a equals b |
| `a != b` | `True` if a is not equal to b |
| `a <= b, a < b` | `True` if a is less than (less than or equal) to b |
| `a > b, a >= b` | `True` if a is greater than (greater than or equal) to b |
| `a is b` | `True` if a and b reference the same Python object |
| `a is not b` | `True` if a and b reference different Python objects |

## Scalar Types (built-in data types)

Python along with its standard library has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and time. These “single value” types are sometimes called ***scalar types***.

*Standard Python scalar types*



| Type | Description |
|---|---|
| `None` | The Python “null” value (only one instance of the `None` object exists) |
| `str` | String type; holds Unicode (UTF-8 encoded) strings |
| `bytes` | Raw ASCII bytes (or Unicode encoded as bytes) |
| `float` | Double-precision (64-bit) floating-point number (note there is no separate `double` type) |
| `bool` | A `True` or `False` value |
| `int` | Arbitrary precision signed integer |

### Numeric types

The primary Python types for numbers are ***int*** and ***float***:

```
In [49]: ival = 17239871
In [50]: ival ** 6
```

```
Out[50]: 262545192910924565969654629132307297011O2721
```

```
In [51]: fval = 7.243
```

```
In [52]: fval2 = 6.78e-5
```

Integer division not resulting in a whole number will always yield a floating-point number:

```
In [53]: 3 / 2
```

```
Out[53]: 1.5
```

To get C-style integer division (which drops the fractional part if the result is not a whole number), use the floor division operator **//**:

```
In [112]: 3 // 2
```

```
Out[112]: 1
```

Another operator available is the modulo (%) operator, which returns the integer remainder of the division. dividend % divisor = remainder.

```
In [113]: 11 % 3
```

```
Out[113]: 2
```

You can also delete the reference to a number object by using the **del** statement.

```
In [131]: var1 = 1
          var2 = 10
```

```
In [133]: del var1, var2
```

## Strings

You can write string literals using either single quotes ' or double quotes ":

```
In [55]: a = 'one way of writing a string'
         b = "another way"
```

For multiline strings with line breaks, you can use triple quotes, either ' ' ' or " " ":

In [56]:
```
c = """
This is a longer string that
spans multiple lines
"""
```

The line breaks after " " " and after lines are included in the string:

In [57]:
```
c.count('\n')
```

Out[57]:
```
3
```

Python strings are immutable; you cannot modify a string:

In [58]:
```
a = 'this is a string'
a[10] = 'f'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-58-cfa170a67205> in <module>()
      1 a = 'this is a string'
----> 2 a[10] = 'f'

TypeError: 'str' object does not support item assignment
```

In [59]:
```
b = a.replace('string', 'longer string')
```

Afer this operation, the variable a is unmodified:

In [60]:
```
a, b
```

Out[60]:
```
('this is a string', 'this is a longer string')
```

Many Python objects can be converted to a string using the str function:

In [61]:
```
a = 5.6
s = str(a)
```

In [62]:
```
print(s)
```

```
5.6
```

Strings are a sequence of Unicode characters and therefore can be treated like other sequences, such as lists and tuples:

In [63]:
```
s = 'python'
```

```
In [64]: list(s)

Out[64]: ['p', 'y', 't', 'h', 'o', 'n']

In [65]: s[:3]

Out[65]: 'pyt'
```

The backslash character **\** is an ***escape*** character, meaning that it is used to specify special characters like newline \n or Unicode characters. To write a string literal with backslashes, you need to escape them:

```
In [66]: s = '12\34'

In [67]: s

Out[67]: '12\x1c'

In [68]: s = "12\\34"

In [69]: s

Out[69]: '12\\34'
```

If you have a string with a lot of backslashes and no special characters, you might find this a bit annoying. Fortunately you can preface the leading quote of the string with **r**, which means that the characters should be interpreted as is:

```
In [70]: s = r'this\has\no\special\characters'

In [71]: s

Out[71]: 'this\\has\\no\\special\\characters'
```

Adding two strings together concatenates them and produces a new string:

```
In [72]: a = 'this is the first half '
         b = 'and this is the second half'

In [134]: a + b

Out[134]: 11
```

In [136]:
```
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
```

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

## String formatting

In [74]:
```
template = '{0:.2f} {1:s} are worth US${2:d}'
```

In [75]:
```
template.format(4.5560, 'Argentine Pesos', 1)
```

Out[75]:
```
'4.56 Argentine Pesos are worth US$1'
```

- {0:.2f} means to format the first argument as a floating-point number with two decimal places.
- {1:s} means to format the second argument as a string.
- {2:d} means to format the third argument as an exact integer.

In [115]:
```
name = "John"
print("Hello, %s!" % name)
```

```
Hello, John!
```

In [116]:
```
name = "John"
age = 23
print("%s is %d years old." % (name, age))
```

```
John is 23 years old.
```

In [118]:
```
mylist = [1,2,3]
print("A list: %s" % mylist)
```

```
A list: [1, 2, 3]
```

- %s - String (or any object with a string representation, like numbers)
- %d - Integers
- %f - Floating point numbers

- % .\<number of digits\> f - Floating point numbers with a fixed amount of digits to the right of the dot.
- %x/%X - Integers in hex representation (lowercase/uppercase)

```
In [ ]:
```

## Bytes and Unicode

In modern Python (i.e., Python 3.0 and up), Unicode has become the first-class string type to enable more consistent handling of ASCII and non-ASCII text. In older ver- sions of Python, strings were all bytes without any explicit Unicode encoding. You could convert to Unicode assuming you knew the character encoding:

```
In [76]: val = "español"
```

```
In [77]: val
```

```
Out[77]: 'español'
```

We can convert this Unicode string to its UTF-8 bytes representation using the encode method:

```
In [78]: val_utf8 = val.encode('utf-8')
```

```
In [79]: val_utf8
```

```
Out[79]: b'espa\xc3\xb1ol'
```

```
In [80]: type(val_utf8)
```

```
Out[80]: bytes
```

Assuming you know the Unicode encoding of a bytes object, you can go back using the decode method:

```
In [81]: val_utf8.decode('utf-8')
```

```
Out[81]: 'español'
```

While it’s become preferred to use UTF-8 for any encoding, for historical reasons you may encounter data in any number of different encodings:

```
In [82]: val.encode('latin1')
```

```
Out[82]: b'espa\xf1ol'
```

In [83]:
```
val.encode('utf-16')
```
Out[83]:
```
b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

In [84]:
```
val.encode('utf-16le')
```
Out[84]:
```
b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

It is most common to encounter bytes objects in the context of working with files, where implicitly decoding all data to Unicode strings may not be desired. Though you may seldom need to do so, you can define your own byte literals by prefixing a string with **b**:

In [85]:
```
bytes_val = b'this is bytes'
bytes_val
```
Out[85]:
```
b'this is bytes'
```

In [86]:
```
decoded = bytes_val.decode('utf8')
decoded # this is str (Unicode) now
```
Out[86]:
```
'this is bytes'
```

## Booleans

In [87]:
```
True and True
```
Out[87]:
```
True
```

In [88]:
```
False or True
```
Out[88]:
```
True
```

## Type casting

The **str**, **bool**, **int**, and **float** types are also functions that can be used to cast values to those types:

In [89]:
```
s = '3.14159'
fval = float(s)
```

In [90]:
```
type(fval)
```
Out[90]:
```
float
```

```
In [91]: int(fval)
Out[91]: 3
```

```
In [92]: bool(fval)
Out[92]: True
```

```
In [93]: bool(0)
Out[93]: False
```

## None

**None** is the Python null value type. If a function does not explicitly return a value, it implicitly returns **None**:

```
In [94]: a = None
         a is None
Out[94]: True
```

```
In [95]: b = 5
         b is not None
Out[95]: True
```

**None** is also a common default value for function arguments:

```
In [96]: def add_and_maybe_multiply(a, b, c=None):
             result = a + b
             if c is not None:
                 result = result * c
             return result
```

**None** is not only a reserved keyword but also a unique instance of **NoneType**:

```
In [97]: type(None)
Out[97]: NoneType
```

## Dates and times

The built-in Python **datetime** module provides **datetime**, **date**, and **time** types. The **datetime** type, combines the information stored in date and time and is the most commonly used:

In [98]:
```
from datetime import datetime, date, time
```

In [99]:
```
dt = datetime(2011, 10, 29, 20, 30, 21)
```

In [100]:
```
dt.day, dt.second, dt.minute
```

Out[100]:
```
(29, 21, 30)
```

Given a **datetime** instance, you can extract the equivalent **date** and **time** objects by calling methods on the **datetime** of the same name:

In [101]:
```
dt.date(), dt.time()
```

Out[101]:
```
(datetime.date(2011, 10, 29), datetime.time(20, 30, 21))
```

The **strftime** method formats a datetime as a string:

In [102]:
```
dt.strftime('%m/%d/%Y %H:%M')
```

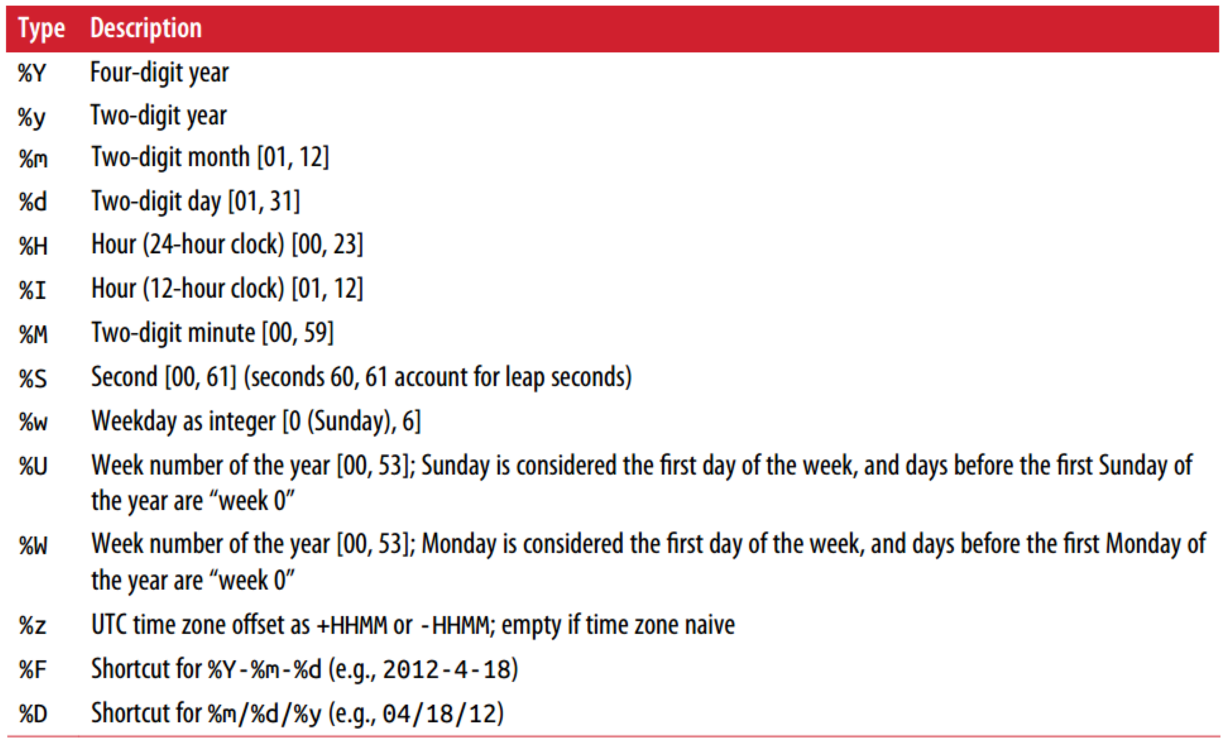
Out[102]:
```
'10/29/2011 20:30'
```

Strings can be converted (parsed) into **datetime** objects with the **strptime** function:

In [103]:
```
datetime.strptime('20091031', '%Y%m%d')
```

Out[103]:
```
datetime.datetime(2009, 10, 31, 0, 0)
```

*Datetime format specification (ISO C89 compatible)*



| Type | Description |
|---|---|
| %Y | Four-digit year |
| %y | Two-digit year |
| %m | Two-digit month [01, 12] |
| %d | Two-digit day [01, 31] |
| %H | Hour (24-hour clock) [00, 23] |
| %I | Hour (12-hour clock) [01, 12] |
| %M | Two-digit minute [00, 59] |
| %S | Second [00, 61] (seconds 60, 61 account for leap seconds) |
| %w | Weekday as integer [0 (Sunday), 6] |
| %U | Week number of the year [00, 53]; Sunday is considered the first day of the week, and days before the first Sunday of the year are "week 0" |
| %W | Week number of the year [00, 53]; Monday is considered the first day of the week, and days before the first Monday of the year are "week 0" |
| %z | UTC time zone offset as +HHMM or -HHMM; empty if time zone naive |
| %F | Shortcut for %Y-%m-%d (e.g., 2012-4-18) |
| %D | Shortcut for %m/%d/%y (e.g., 04/18/12) |

Replacing time fields of a series of datetimes:

```
In [104]: dt.replace(minute=0, second=0)
```

```
Out[104]: datetime.datetime(2011, 10, 29, 20, 0)
```

The difference of two **datetime** objects produces a **datetime.timedelta** type:

```
In [105]: dt2 = datetime(2011, 11, 15, 22, 30)
```

```
In [106]: delta = dt2 - dt
```

```
In [109]: delta , type(delta)
```

```
Out[109]: (datetime.timedelta(17, 7179), datetime.timedelta)
```

The output **timedelta(17, 7179)** indicates that the timedelta encodes an offset of 17 days and 7,179 seconds.

Adding a **timedelta** to a **datetime** produces a new shifted datetime:

```
In [108]: dt, dt + delta
```

```
Out[108]: (datetime.datetime(2011, 10, 29, 20, 30, 21),
           datetime.datetime(2011, 11, 15, 22, 30))
```

## Types of Operator

Arithmetic Operators

Comparison (Relational) Operators

Assignment Operators

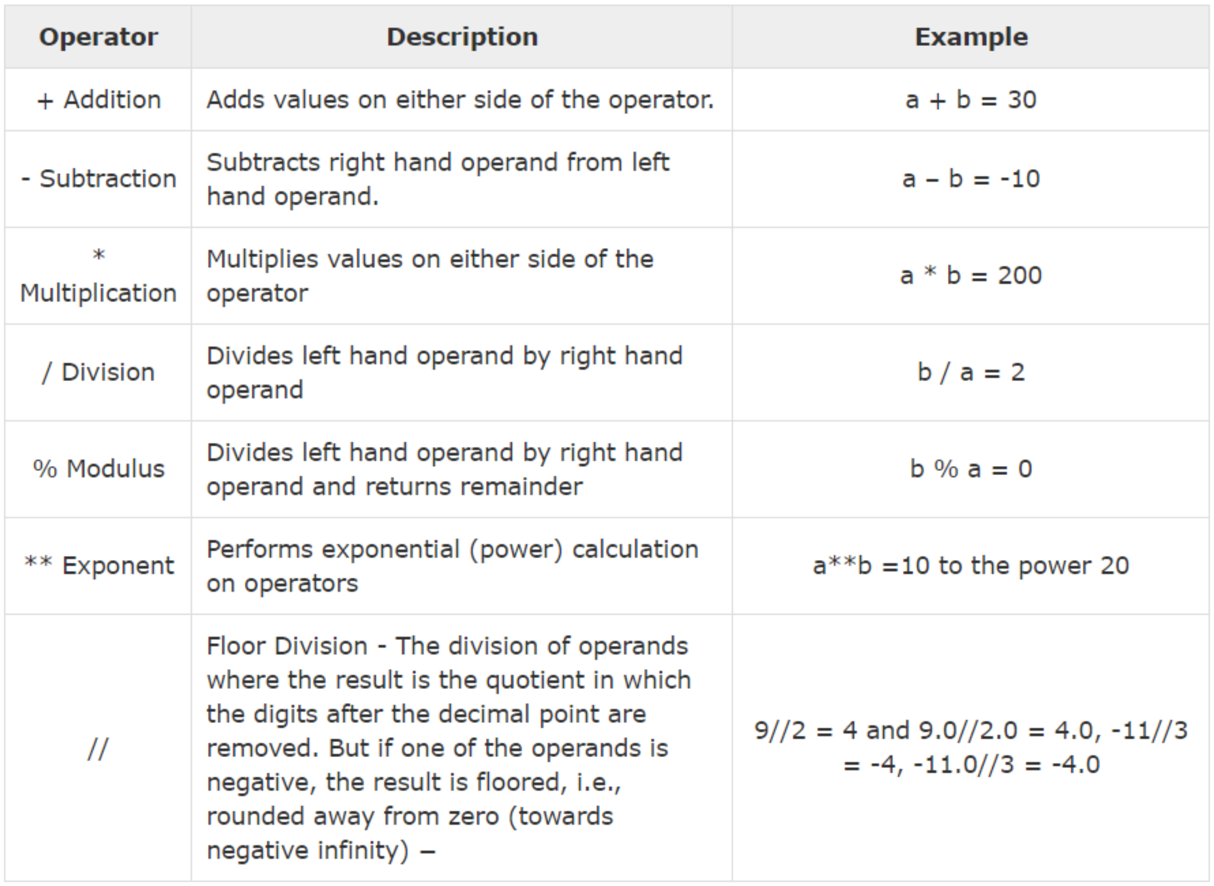
Logical Operators

Bitwise Operators

Membership Operators

Identity Operators

### Python Arithmetic Operators

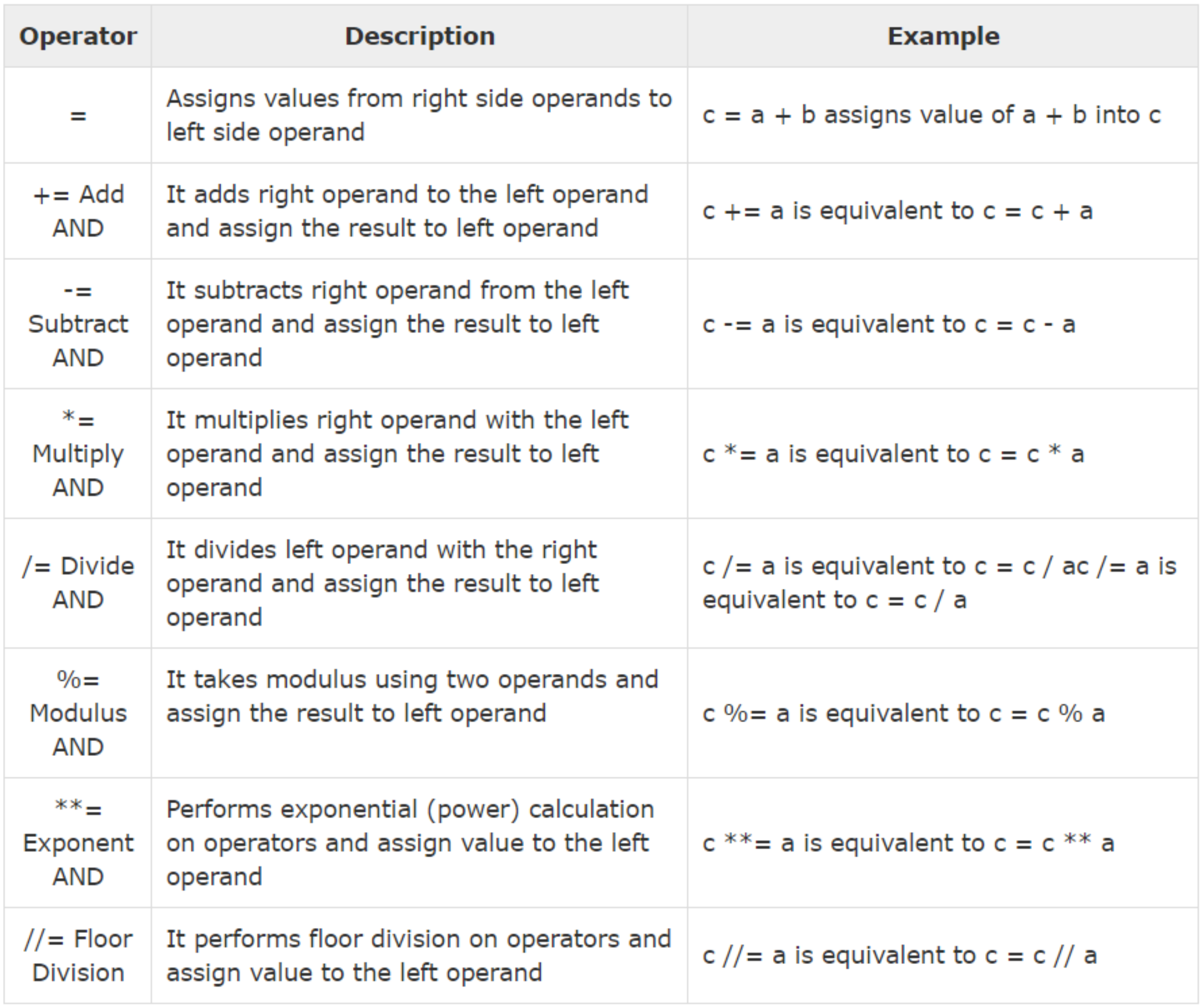


| Operator | Description | Example |
|---|---|---|
| + Addition | Adds values on either side of the operator. | a + b = 30 |
| - Subtraction | Subtracts right hand operand from left hand operand. | a – b = -10 |
| * Multiplication | Multiplies values on either side of the operator | a * b = 200 |
| / Division | Divides left hand operand by right hand operand | b / a = 2 |
| % Modulus | Divides left hand operand by right hand operand and returns remainder | b % a = 0 |
| ** Exponent | Performs exponential (power) calculation on operators | a**b =10 to the power 20 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) – | 9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0 |

### Python Comparison Operators

| Operator | Description | Example |
|---|---|---|
| == | 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. |
| <= | If the value of left operand is less than or equal to the value of right operand, then condition becomes true. | (a <= b) is true. |

### Python Assignment Operators



| 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 |
| -= 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 / ac /= 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 |

### Python Bitwise Operators

| Operator | Description | Example |
|---|---|---|
| & Binary AND | Operator copies a bit to the result if it exists in both operands | (a & b) (means 0000 1100) |
| \| Binary OR | It copies a bit if it exists in either operand. | (a \| b) = 61 (means 0011 1101) |
| ^ Binary XOR | It copies the bit if it is set in one operand but not both. | (a ^ b) = 49 (means 0011 0001) |
| ~ Binary Ones Complement | It is unary and has the effect of 'flipping' bits. | (~a ) = -61 (means 1100 0011 in 2's complement form due to a signed binary number. |
| << Binary Left Shift | The left operands value is moved left by the number of bits specified by the right operand. | a << 2 = 240 (means 1111 0000) |
| >> Binary Right Shift | The left operands value is moved right by the number of bits specified by the right operand. | a >> 2 = 15 (means 0000 1111) |

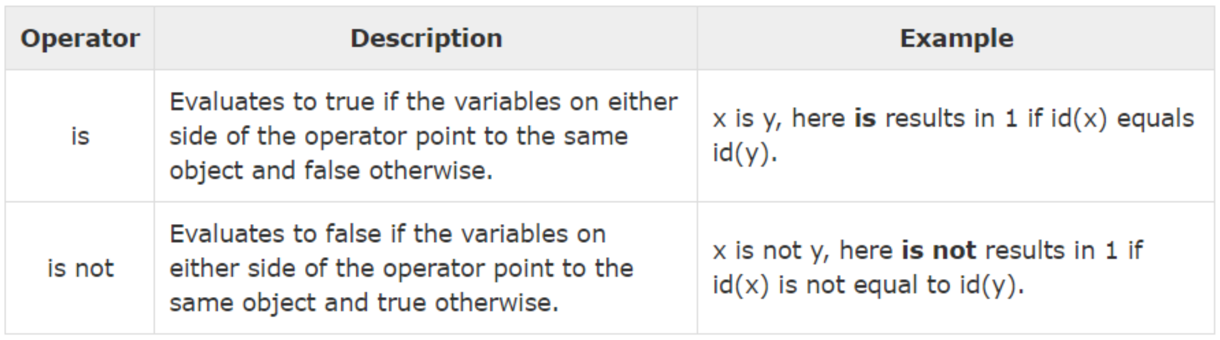
### Python Logical Operators

| 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. |

### 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. |

### Python Identity Operators



| 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). |

Type *Markdown* and LaTeX: $\alpha^2$