Table of contents (../toc.ipynb)

Python semantics

Remember that semantics is the meaning of the language.

Variables are actually pointers

The next slides of topic (variables are actually pointers) are a condensed version of notebook 03 from *A Whirlwind Tour of Python* [VanderPlas2016] (../references.bib), which is under CC0 license.

Assigning a variable in Python is very easy, just use the equal sign:

```
my_var = 7
```

Comparision with C

However, in contrast to other languages like C, the above line of code should be read like:

`my_var` points to a memory bucket which contains currently an integer of seven.

This is very different from C where a similar line of code would be

```c
int my_var = 7;
```

This C code line could be read as:

A container called `my_var` is defined to store integers and it contains currently seven.

Consequences

Because Python variables just point to some object in the memory:

- there is no need to declare a variable type
- the variable type may change
- and this is the reason why Python is called dynamically typed language

Hence you can do things like this in Python which won't work in statically typed languages like C.

In [93]:

```python
my_var = 7  # integer
my_var = 7.1  # float
my_var = "Some string"  # string
my_var = [0, 1, "a list with a string"]  # list with intergers and a string
```

Variables as pointers in practice

Because variables are actually pointers, you might wonder how this code is interpreted.

In [94]:

```
x = [0, 1, 2, 3]
y = x
print(y)
```

```
[0, 1, 2, 3]
```

In [95]:

```
x[3] = 99
print(y)
```

```
[0, 1, 2, 99]
```

Here, the last entry of x was changed and because y point to x, the print y command showed the changed values.

However, if we change the bucket where x points to something different, y still points to the "old" bucket.

In [96]:

```
x = 77.7
print(y)
```

```
[0, 1, 2, 99]
```

Is this safe?

You might wonder if this will cause trouble in equations. But it is safe because:

- Numbers, strings and other basic types are immutable
- This means you can not change their value. You can only change what values the variable points to.

In [97]:

```
x = 10
y = x
# Here actually the variable is changed so that it points to another integer which is 15. Hence, y does not change.
x += 5
print("x =", x)
print("y =", y)
```

```
x = 15
y = 10
```

Python loves objects!

Because there is no need to define the type of a variable it is often said that Python is type-free.

This is wrong!

If no type is given, the Python interpreter selects a type and we can read the type.

```
In [98]:  x = 7
          type(x)

Out[98]:  int
```

```
In [99]:  x = [0, 1, 2, "string"]
          type(x)

Out[99]:  list
```

```
In [100]: x = {"a": 3, "b": 4.4}
          type(x)

Out[100]: dict
```

Hence, Python has types and the type is not linked to the variable but to the object.

You have seen here some of the basic types like `int` (integer), `list`, and `dict` (dictionary).

Everything is an object

You can access different properties of objects with the period `.`

The last line of code set x as pointer to a dictionary and the keys of a dictionary can be accessed with `.keys()` method.

In [101]:

```
x = {"a": 3, "b": 4.4}
x.keys()
```

Out[101]:

```
dict_keys(['a', 'b'])
```

Also very basic objects like integers have attributes and methods.

In [1]:

```
x = 9
print(x.real)   # real attribute of x
print(x.bit_length())   # compute bitlength of x with .bitlenght() method
```

```
9
4
```

Operators

Python's operators can be categorized into:

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

Arithmetic operators

The arithmetic operators in Python are:

Operator	Description	Code example
Addition	Sum of two variables	`a + b`
Subtraction	Difference of two variables	`a - b`
Multiplication	Product of two variables	`a * b`
Exponentiation	-	`a ** b`
Division	Quotient of two variables	`a / b`
Floor division	Quotient without fractional part	`a // b`
Modulus	Returns integer which remains after division	`a % b`
Negation	Negative of variable	`-a`
Matrix product	Introduced in Python 3.5, requires numpy	`a @ b`

In [2]:

```
# here some operators in combination
((5 + 3) / 4) ** 2
```

Out[2]:

```
4.0
```

In [3]:

```
# true division
21 / 2
```

Out[3]:

```
10.5
```

In [4]:

```
# floor division
21 // 2
```

Out[4]:

```
10
```

In [5]:

```
# modulus
21 % 2
```

Out[5]:

```
1
```

Comparison operators

The comparison operators in Python return a bool (`True` or `False`) and these operators are:

Code example	Description
`a == b`	a equal b
`a != b`	a not equal b
`a < b`	a less than b
`a <= b`	a less or equal b
`a > b`	a greater than b
`a >= b`	a greater or equal b

In [6]:

```
a = 1
b = 5
a < b
```

Out[6]:

```
True
```

In [7]:

```
a != b
```

Out[7]:

```
True
```

In [8]:

```
a > b
```

Out[8]:

```
False
```

In [11]:

```
# check floor division
25 // 2 == 12
```

Out[11]:

```
True
```

In [13]:

```
# 30 is between 24 and 50
24 < 30 < 50
```

Out[13]:

```
True
```

Assignment operators

Beside the simple assignment $=$, there are:

Code example	Description
`a += 5`	Add And
`a -= 5`	Subtract And
`a *= 5`	Multiply And
`a **= 5`	Exponent And
`a /= 5`	Division And
`a %= 5`	Modulus And
`a //= 5`	Floor division And

In [22]:

```
a = 5
a += 5
print(a)
```

```
10
```

In [23]:

```
a -= 10
print(a)
```

```
0
```

Logical operators

These operators are designed for boolean variables. There are logical `and` , `or` , `not` . The operator `xor` is missing but can be constructed.

```
In [24]: a = True
         b = False
         a and b

Out[24]: False
```

```
In [25]: a or b

Out[25]: True
```

```
In [29]: not(a, b)

Out[29]: False
```

Bitwise operators

- These operators are rather advanced and barely used for standard tasks.
- They compare the binary representation of numbers, which can be accessed with `bin()` function.

```
In [38]: bin(10)
```

```
Out[38]: '0b1010'
```

Which is read as: `0b` binary format, $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 10$

Fore sake of completeness the operators are:

- Bitwise AND `a & b`
- Bitwise OR `a | b`
- Bitwise XOR `a ^ b`
- Bitshift left `a << b`
- Bitshift right `a >> b`
- Binary complement (flipping the bit) `~a`

Membership operators

The membership operators are designed to find values in lists, tuples or strings.

The two operators are `in` and `not in` and they return `True` or `False`.

```
In [30]: a = 2
         b = [0, 1, 2, 3, 4]
         a in b

Out[30]: True
```

```
In [32]: c = 8
         c not in b

Out[32]: True
```

```
In [33]: strg = "hi here is a text"
         "text" in strg

Out[33]: True
```

Identity operators

They compare memory location of objects and are called `is` and `is not`.

```
In [35]: a = 5
         b = 2
         a is b
```

```
Out[35]: False
```

```
In [37]: a = 10
         b = a
         a is b
```

```
Out[37]: True
```