

Python uses Dynamic Typing. This means you can reassign variables to different data types. This makes Python very flexible in assigning data types this is different than other languages that are “Statically Typed”. Python uses Dynamic Typing The type of a variable can change during the execution of a script. Or to be precise: A new object, which can be of any type, will be assigned to it. We illustrate this in our following example:

i = 42 # data type is implicitly set to integer

i = 42 + 0.11 # data type is changed to float

i = "forty" # and now it will be a string

i = [“muni”,”swamy”]

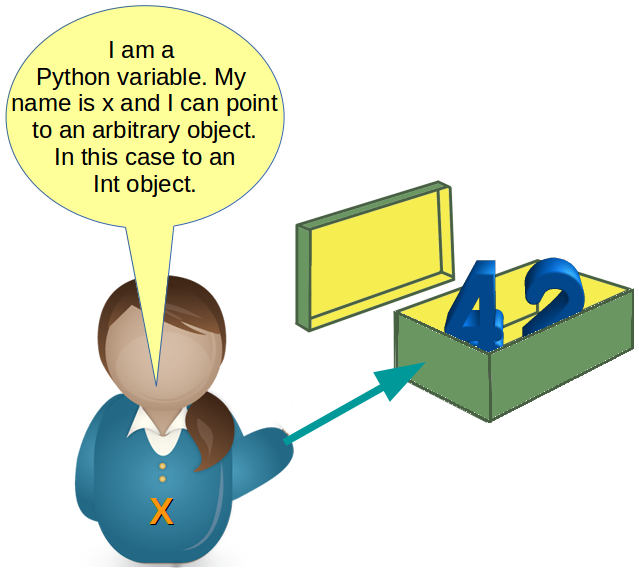
**Pros:**

* Very easy to work with
* Faster development time

**Cons:**

* May result in bugs for unexpected data types.
* You need to be aware of **type()**

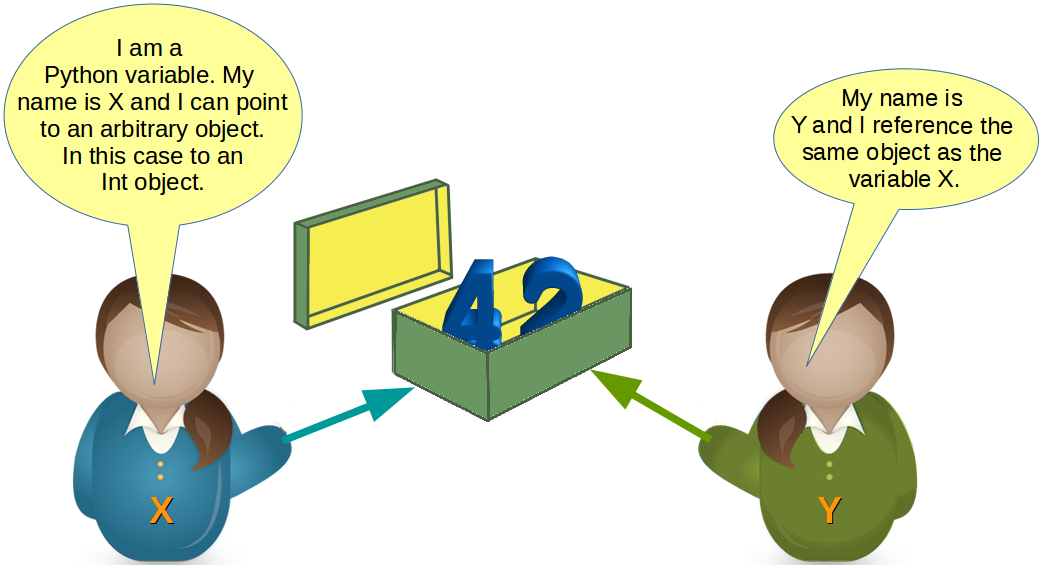
Python automatically takes care of the physical representation for the different data types, i.e. an integer values will be stored in a different memory location than a float or a string.



As variables are pointing to objects and objects can be of arbitrary data type, variables cannot have types associated with them. This is a huge difference to C, C++ or Java, where a variable is associated with a fixed data type.

>>> x = 42

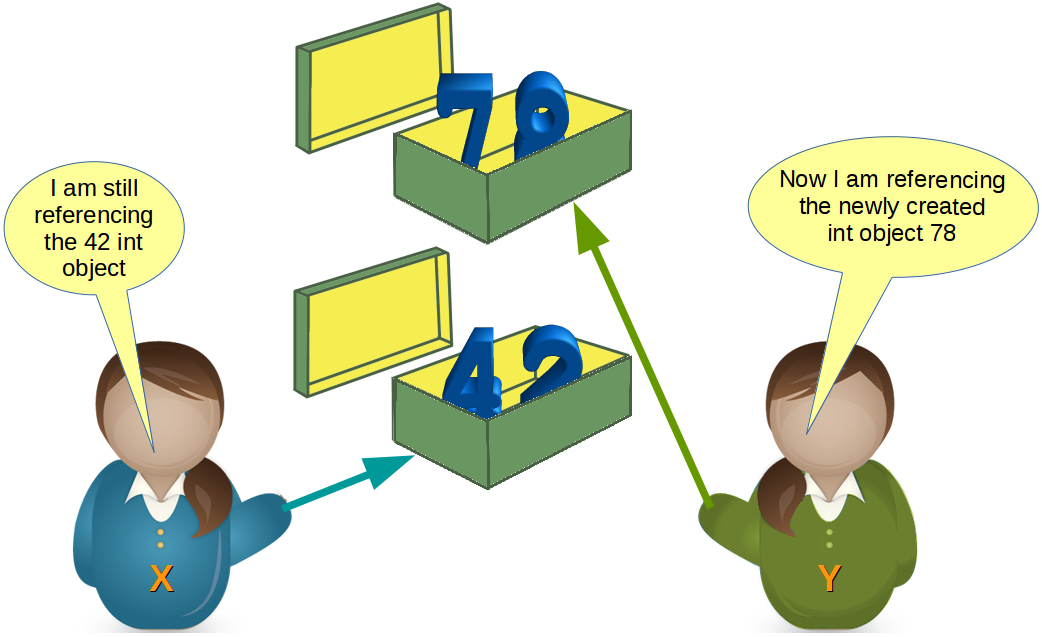
>>> y = x



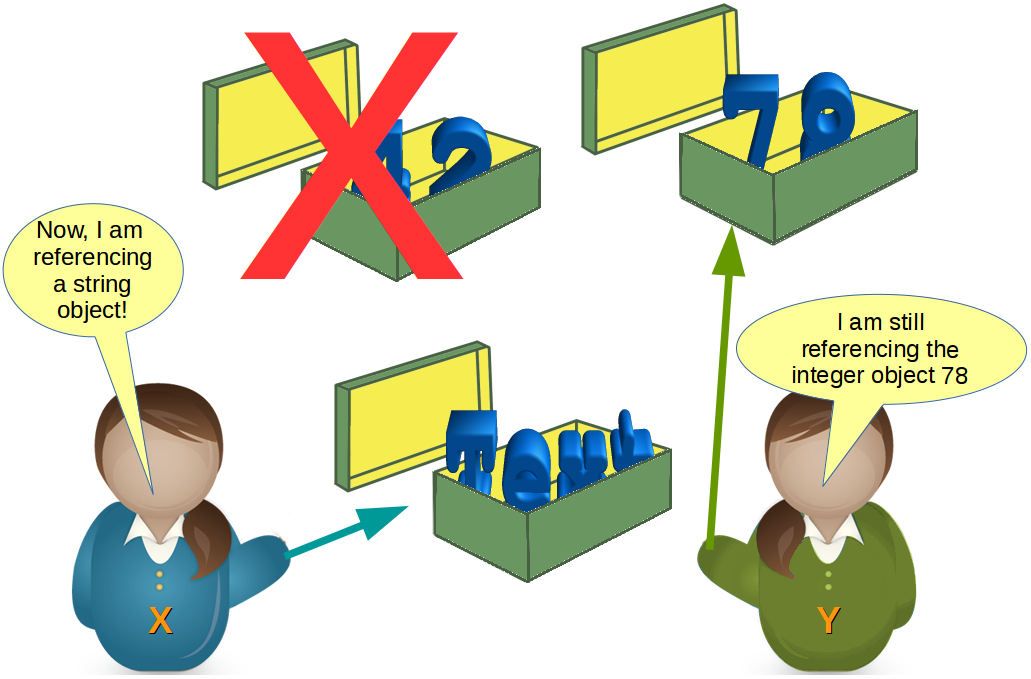
What will happen, when we execute

y = 78

Python will create a new integer object with the content 78 and then the variable y will reference this newly created object, as we can see in the following picture:



Most probably, we will see further changes to the variables in the flow of our program. There might be, for example, a string assignment to the variable x. The previously integer object "42" will be orphaned after this assignment. It will be removed by Python, because no other variable is referencing it.



### id Function

You may ask yourself, how can we see or prove that x and y really reference the same object after the assignment y = x of our previous example?

The identity function id() can be used for this purpose. Every instance (object or variable) has an identity, i.e. an integer which is unique within the script or program, i.e. other objects have different identities.

So, let's have a look at our previous example and how the identities will change:

>>> x = 42

>>> id(x)

10107136

>>> y = x

>>> id(x), id(y)

(10107136, 10107136)

>>> y = 78

>>> id(x), id(y)

(10107136, 10108288)

>>>

### Valid Variable Names

A Python identifier is a name used to identify a variable, uppercase letters "A" to "Z", the lowercase letters "a" through "z", the digits 0 through 9 except for the first character. There can be no spaces in the name, use \_ instead. Python 3.x is based on Unicode. This means that variable names and identifier names can additionally contain Unicode characters as well. Identifiers are unlimited in length. Case is significant. The fact that identifier names are case-sensitive can cause problems to some Windows users, where file names are case-insensitive, for example.

Exceptions from the rules above are the special Python keywords, as they are described in the following paragraph.

The following variable definitions are all valid:

>>> height = 10

>>> maximum\_height = 100

>>>

>>> υψος = 10

>>> μεγιστη\_υψος = 100

>>>

>>> MinimumHeight = 1

### Python Keywords

You can get the list of Python keywords in the interactive shell by using help. You type in help() in the interactive, but please don't forget the parenthesis:

>>> help()

help>

help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

False def if raise

None del import return

True elif in try

and else is while

as except lambda with

assert finally nonlocal yield

break for not

class from or

continue global pass

help>

# String:

[start:stop:step]

Start: is the numerical index for the slice start.

Stop: is the numerical index for the slice stop.

Step: is the size of “jump” you take.

name = *"muni"*;

print(name); # muni

name1 = *'swamy'*;

print(name1); # swamy

x = *"abcdefghijklmno"*;

print(x[3:7]); # defg

print(x[3:7:2]); # df

print(x[3]); # d

print(x[0]); # a

print(x[-1]); # o

print(x[-2]); # n

print(x[::]); # abcdefghijklmno

print(x[::3]); # adgjm

print(x[::-1]); # onmlkjihgfedcba (String reverse)

print(*"this is {} {} {}"*.format(*"muni"*, *"swamy"*, *"palla"*)); # this is muni swamy palla

print(*"lenth "*,len(x)); # lenth 15

Strings are immutable! (meaning you can't use indexing to change individual elements of a string).

New in Python 3.6, f-string.

name = *"muni"*;

print(*f"name is {name}"*); # name is muni

# Float:

Float formatting follows “{value:width.precision f}”

fvalue = 100/177;

print(*"value is {:1.3f}"*.format(fvalue)); # value is 0.565

# List:

list1 = [*'one'*, *'two'*, *'three'*];

list2 = [*'four'*, *'five'*];

list3 = list1 + list2;

print(list3) # ['one', 'two', 'three', 'four', 'five']

list1 = [*'one'*, *'two'*, *'three'*];

list1.append(*'four'*) ; # Adding element at the end.

print(list1); # ['one', 'two', 'three', 'four']

list1 = [*'one'*, *'two'*, *'three'*];

list1.pop(); # Removing element at the end.

print(list1); # ['one', 'two']

list1 = [*'one'*, *'two'*, *'three'*];

list1.pop(1); # Removing element at position 1. By default value is -1, that means last element in the list.

print(list1); # ['one', 'three']

list1 = [*'d'*,*'a'*,*'c'*,*'b'*];

list1.sort();

print(list1) # ['a', 'b', 'c', 'd']

list1 = [*'d'*,*'a'*,*'c'*,*'b'*];

#list1.sort(key=None, reverse=True); # Reverse order

list1.sort(reverse=True); # Reverse order

print(list1) # ['d', 'c', 'b', 'a']

# Dictionaries:

Dictionaries use curly braces and colons to signify the keys and their associated values.

list1 = [*'k1'*, *'k2'*, *'k3'*];

print(list1[0]); # k1

dictionary = {*'k1'*:1, *'k2'*:2, *'k3'*:3, *'k4'*:4};

print(dictionary[*'k3'*]); # 3

dictionary = {*'k1'*:1, *'k2'*:[2,22,222]};

print(dictionary);

dictionary = {*'k1'*:1, *'k2'*:[2,22,222]};

dictionary[*'k2'*][0]=2222

print(dictionary);

# Tuples:

Tuples are very similar to lists. However, they have one key difference – **Immutability**.

Once element inside a tuple, it can’t be reassigned. Tuples uses parenthesis (1, 2, 3).

list1 = [10, 20, 30, 40, 50];

tuple1 = (10, 20, 30, 40, 30);

print(len(tuple1)); # 5

print(tuple1.index(30)); # 2

print(tuple1.count(30)); # 2

tuple1[0] = 60; # not allowed for tuples

list1[0] = 60; # allowed for list

# Sets:

Sets are unordered collections of **unique** elements.

set1 = set();

set1.add(10);

set1.add(20);

print(set1); # {10, 20} similar to dictionary data store

list1 = [1,1,1,1,2,2,2,2];

print(set(list1)); # {1, 2}

# Files:

file1 = open(*'test.txt'*);

print(file1.read()); # prints file data.

file1.seek(0); # move cursor to beginning of the file.

print(file1.read()); # prints file data.

file1.close(); # to close file.

**With:**

You can also work with file objects using the with statement. It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it’s good practice to use the with statement where applicable.

One bonus of using this method is that any files opened will be closed automatically after you are done. This leaves less to worry about during clean-up.

Syntax:

***with open(“filename”) as file:***

with open(*'test.txt'*, mode=*'r'*) as new\_file:

print(new\_file.read());

**Reading, Writing, Appending Modes:**

**mode=’r’** is read only

**mode=’w’** is write only (will override files or create new)

**mode=’a’** is append only (will add on to files)

**mode=’r+’** is reading and writing

**mode=’w+’** is writing and reading (Overwrites existing files or create a new file)