Section 2 Expressions, Variables and Objects

There is one famous saying: *Everything is an object in Python!*

In Python, each object has

- an identity,
- a type, and
- a value

Identity and id( )

Roughly speaking, the id( ) function returns an integer called identity, representing the unique memory address of an object.

In [38]:

```
print(id(3.0))
print(id(3)) # integer 3 has an identity
```

```
1995275930736
140712428840816
```

In [4]:

```
print(id(5.0)) # float 5 has different identity
print(id(5))
```

```
1995275928720
140712428840880
```

In [20]:

```
print(id("python"))# string 'python' has an identity. Btw, there is no difference betwe
print(id('python'))
```

```
1995244205872
1995244205872
```

In [9]:

```
id([1,2,3]) # list [1,2,3] has another identity
```

Out[9]:

```
1995275580864
```

In [11]:

```
id(abs) # built-in function abs also has an unique indentity!
```

Out[11]:

```
1995202823168
```

In [12]:

```
a = 3
id(a) #exactly the same as id(3) !!!
```

Out[12]:

```
140712428840816
```

Type and type( )

Below are the common built-in types of Python. We're going to define our own types later using Class in Python. Popular data science packages also define their own types.

In [13]:

```
type(3)
```

Out[13]:

```
int
```

In [16]:

```
print(type(True))
print(type(3==5))
```

```
<class 'bool'>
<class 'bool'>
```

In [10]:

```
type(5.)
```

Out[10]:

```
float
```

In [10]:

```
type('python')
```

Out[10]:

```
str
```

In [18]:

```
type([1,2,3.7,'hey'])
```

Out[18]:

```
list
```

In [12]:

```
type(abs)
```

Out[12]:

```
builtin_function_or_method
```

Expression, Variable, Value and Object

Compared with the concept of *object*, perhaps you're more familiar with the notion of *variables* and *values* in Matlab. With the assignment operators (=), you can assign the *values* to *variables* through expressions in Matlab.

Formally, similar things happen in Python.

In [19]:

```
string = 'python'

print(id(string))

type(string)
```

```
1995244205872
```

Out[19]:

```
str
```

Below we're going to develop a deep understanding of what happens after executing the expression **variable = value** in Python -- dig deep into your computer memory space!

The basic conclusion can be stated as follows: **In Python, variables are just the references to objects.**

Instead of saying that we *assign values to variables* in python, perhaps it's more rigorous to say that *we use variables to point toward objects with certain values*.

In fact, it is even not the most accurate way to use the word "variables". The more appropriate word in Python might be "names" or "identifiers".

In [33]:

```
a = 3
print(id(a))
a = 1
print(id(a))
```

```
140712428840816
140712428840752
```

In [2]:

```
print(id(1000))
a = 1000 # creating an int object with value 1000, and use variable a as the reference
print(id(a))
c = 1000
print(id(c))
b = a  # link the SAME object to b
print(id(b))
```

```
2295892837136
2295892836720
2295892837264
2295892836720
```

In [37]:

```
a = 1000 # creating an int object with value 1000, and use variable a as the reference
print(id(a))
b = a # link the SAME object to b -- now a and b refers to exactly the same object !
print(id(b))
b = 1 # creating a new int object with value 1, and use variable b as the reference
print(id(b))
```

```
1995275927920
1995275927920
140712428840752
```

The rules for immutable objects are slightly different for large integers and for floating point numbers. In 2 of the 3 following examples, you will notice the same immutable object occupying multiple memory addresses.

If a memory address can no longer be referenced, sometimes Python will clear that memory address. Without this, it would be possible to fill the memory by assigning too many variables.

This is why `print(id(1000))` does not match `print(id(a))` in the above. This is also why the memory addresses can change when we rerun a cell.

In [14]:

```
print(id(256))
a = 256 # memory rules for small integers
print(id(a))
c = 256
```

```
print(id(c))
b = a  # link the SAME object to b
print(id(b))
```

```
140718362412816
140718362412816
140718362412816
140718362412816
```

In [13]:

```
print(id(257))
a = 257 # memory rules for large integers
print(id(a))
c = 257
print(id(c)) #Two versions of an immutable object, at two memory addresses
b = a  # link the SAME object to b
print(id(b))
```

```
2295892836912
2295892837424
2295892837904
2295892837424
```

In [15]:

```
print(id(2.1))
a = 2.1 # memory rules for floating point numbers
print(id(a))
c = 2.1
print(id(c)) #Two versions of an immutable object, at two memory addresses
b = a  # link the SAME object to b
print(id(b))
```

```
2295891973904
2295892208464
2295892836752
2295892208464
```

In [ ]: