python_lecture_01_data_types

January 29, 2020

1 Programming with Python

2 1 Basic data types

[1]:
```
# <-- This hash tag is the comment symbol in python.
# the python interpreter ignores the whole line starting by this symbol.
# It is used in your program to comment on code
```

2.1 1.1 Strings

String literals

In computer programming, a **string** is a **sequence of charaters**. Strings literals are enclosed in **delimiters**. In python you can choose between two delimiters: **single quotes** and **double quotes**.

[2]:
```
'Hello World'
```

[2]:
```
'Hello World'
```

[3]:
```
"Hello World in double quotes!"
```

[3]:
```
'Hello World in double quotes!'
```

A string may be **empty**, i.e. consist of 0 characters:

[4]:
```
''
```

[4]:
```
''
```

Obviously, a string literal cannot contain its delimiter:

[5]:
```
'Land's End'
```

```
  File "<ipython-input-5-14bc6f4d08f6>", line 1
    'Land's End'
          ^
SyntaxError: invalid syntax
```

This issue can be avoided by using the other delimiter:

[6]:
```
"Land's End"
```

[6]:
```
"Land's End"
```

[7]:
```
'"Right" said fred.'
```

[7]:
```
'"Right" said fred.'
```

However, there are string which contain both delimiters:

[8]:
```
'"Right" is Fred's motto.'
```

```
  File "<ipython-input-8-f03ea044d964>", line 1
    '"Right" is Fred's motto.'
                      ^
SyntaxError: invalid syntax
```

[9]:
```
"Right" is Fred's motto."
```

```
  File "<ipython-input-9-4a8d1661e502>", line 1
    "Right" is Fred's motto."
                             ^
SyntaxError: EOL while scanning string literal
```

In such cases you need to tell the computer he should not interpret a quote within a string as a quote but as a **metacharacter**. This can be achieved with an **escape sequence**. Escape sequences start with an **escape character ** an consist of this escape characted and the character that** immediately follows it**:

[10]:
```
"\"Right\" is Fred's motto."  # use \' to escape in-between quote signs ...
```

[10]:
```
'"Right" is Fred\'s motto.'
```

[11]:
```
'"Right" is Fred\'s motto.'
```

[11]:
```
'"Right" is Fred\'s motto.'
```

If you want to use in a string and not as a escape sign, you can use a raw string

[12]:
```
print('C:\some\name') # \n is a command for new line
```

```
C:\some
ame
```

[13]:
```
print(r'C:\some\name')
```

```
C:\some\name
```

2.1.1 Printing a string

We can print out strings by *simply* calling the print function:

[14]:
```
# Call print function and pass in a string as argument
print('"Right" is Fred\'s motto.')
```

```
"Right" is Fred's motto.
```

Within a Jupyter Notebook, the print function simply prints the string that we pass in to the screen. In contrast to a *return value*, the print function applies some pretty formating:

[15]:
```
# print '"Right" is Fred\'s motto.'...
print('"Right" is Fred\'s motto.')
# ... and get the return value of a string literal
'"Right" is Fred\'s motto.'
```

```
"Right" is Fred's motto.
```

[15]:
```
'"Right" is Fred\'s motto.'
```

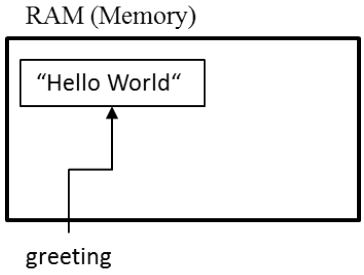
[ ]:
```

```

2.1.2 Excursion: persisting data in computer memory

So far, all string literals that we created were **volatile**, i.e. they vanished from computer memory right after they were printed on screen. In order to persist strings, or any other kind of data in computer memory, we need to create a **variable**. Variables consist of an **in-memory storage location**, an **identifier** and of the data to be stored (**value**). As python takes automatically care of the required storage location, all you need to do is to choose an identifier and to **assign** the value to it:

[16]:
```
# assigning a string to an identifier creates a variable
# python's assign operator is the equal sign (=)
greeting = "Hello World"
```

Please note that creating a variable does not have a return value.



Assign a string to a variable

[17]:
```
greeting
```

[17]:
```
'Hello World'
```

2.1.3 Accessing characters in a string

We can access

- **individual** characters using **indexing** [i] and
- a **range of characters** using **slicing** [i:j].

Index starts from 0.

[18]:
```
# assign a string to a variable
name = 'Monty Python'

# get first character
# remeber that indexing starts at 0
name[0]
```

[18]:
```
'M'
```

[19]:
```
# get third character
name[2]
```

[19]:
```
'n'
```

Note that the character at the **starting index *i* is included**, while the character at the **stopping index *j* is not included**. The reason for this behavior is that ***j-i* equals the length** of the returned substring.

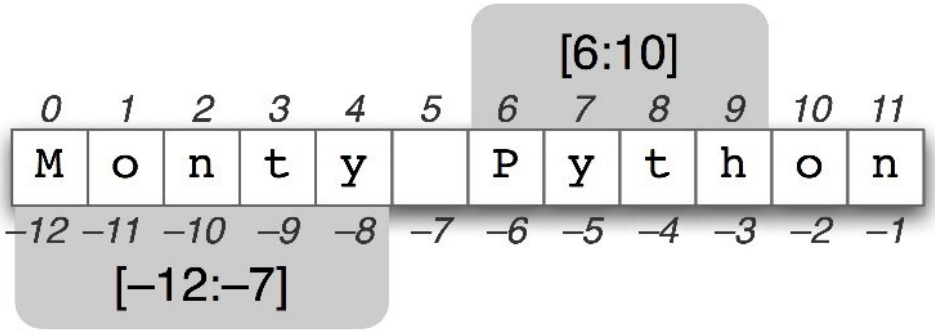
[20]:
```
# get 6th to 9th character using a slice
name[6:10]
```

[20]:
```
'Pyth'
```

Using **negative indices** you can index characters starting from the end of the string:

[21]:
```
name[-12:-7]
```

[21]:
```
'Monty'
```



String indexing in python

Ommitting the index *i*, i.e. providing [:j] to a string, means: "Everything from the beginning to j-1":

[22]:
```
name[:5]
```

[22]:
```
'Monty'
```

[23]:
```
# the above is therefore equivalent to
name[0:5]
```

[23]:
```
'Monty'
```

Correspondingly, ommiting the stopping index, i.e. [i:] means "Everything from i to the end of the string":

[24]:
```
# name = 'Monty Python'
name[6:]
```

[24]:
```
'Python'
```

As the stopping index is not included, this is **not equivalent** to:

[25]:
```
name[6:-1]
```

[25]:
```
'Pytho'
```

Trying to access index out of the range or using a decimal number, an error will be raised:

[26]:
```
name[14]
```

```
␣
↪---------------------------------------------------------------------------

IndexError                                Traceback (most recent call␣
↪last)

<ipython-input-26-793b086a569f> in <module>
----> 1 name[14]

IndexError: string index out of range
```

[27]:
```
name[0.7]
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-27-58ae36dd8b4a> in <module>
----> 1 name[0.7]
```

```
TypeError: string indices must be integers
```

2.1.4 Modifying strings

In python, strings are **immutable**, i.e. they cannot be modified. Therefore, the following does not work:

[28]:
```
# name = 'Monty Python'
name[0] = 'm'
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-28-b0956fdb7b31> in <module>
  1 # name = 'Monty Python'
----> 2 name[0] = 'm'


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

If a string needs to get modified, create a new string:

[29]:
```
'm' + name[1:]
```

[29]:
```
'monty Python'
```

As the right side of an expression is evalauted fully **before** any variable assignments, newly generated strings can be **re-assigned** to their original identifier in one single statement:

[30]:
```
name = 'm' + name[1:]
print(name) # needs to be printed as assignment statements don't have return␣
 ↪values
```

```
monty Python
```

[31]:
```
name
```

[31]:
```
'monty Python'
```

[ ]:
```

```

2.1.5 Deleting strings (or any other python object)

In-memory python objects get destroyed by python's (internal) garbage collection (GC). GC mainly works by counting the number of 'references' an object has. In the most simple case, each object has exactly one reference: its identifier.

[32]:
```
a = 'Some string'
```

[33]:
```
a
```

[33]:
```
'Some string'
```

Now, the in-memory object 'Some string' has exaclty one reference, i.e. *a*.

If you delete *a*, the number of references to 'Some string' gets decremented to 0. When an object's reference counter reaches 0, it gets destroyed by GC.

[34]:
```
# delete the only identifier of 'Some string'
del(a)
# as the reference count of 'Some string' has reached 0, it gets GCed.
# calling a now raises an error
a
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-34-735bd657d797> in <module>
      3 # as the reference count of 'Some string' has reached 0, it gets␣
↪GCed.
      4 # calling a now raises an error
----> 5 a

NameError: name 'a' is not defined
```

2.1.6 Copying a string (or aliasing?)

Consider the following code:

[35]:
```
a = 'hans'
b = a
```

[36]:
```
a
```

[36]:
```
'hans'
```

[37]:
```
b
```

[37]:
```
'hans'
```

[38]:
```
a = 'klaus'
```

[39]:
```
a
```

[39]:
```
'klaus'
```

[40]:
```
b
```

[40]:
```
'hans'
```

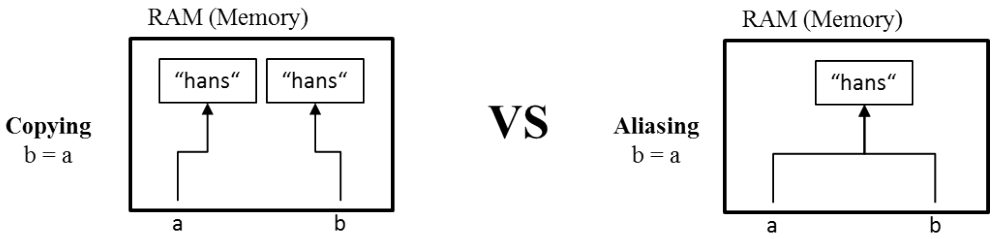
So what will python do, if we assign an existing variable to a new identifier? The answer is: it depends! Theoretically, there are two options: It could either

- create new object with the same value of a, i.e. 'aa', and assign that duplicate to b (copying)

or it could

- create a new identifier b that points to the same in-memory object as a (aliasing)

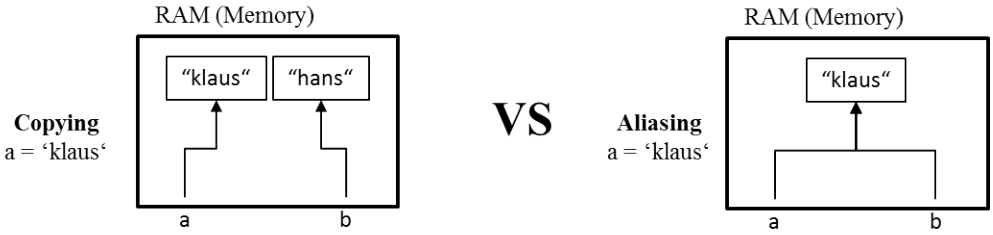
So, what is python doing?



Copy vs alias

Actually that's easy to find out:

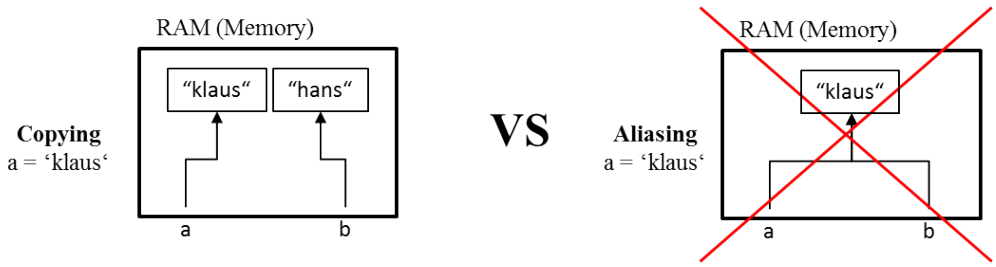
[41]:
```
# reassign a
a = 'klaus'
```



Copy vs alias

[42]:
```
# get b
b
```

[42]:
```
'hans'
```



Copy vs alias

Obviously, python **copied** a to b when we provided it with the $b = a$ statement. If it had **aliased** a to b, changing the value of a would have also changed the output of calling b (as after aliasing, both would have pointed to the same in-memory object).

2.2 1.2 Integers and Floats

Integers are whole numbers, like 5, 0, -5. Floats are decimal numbers, like 1.2, 5.0 In Python, the decimal separator is a dot.

[43]:
```
print('Examples of valid python number literals:')
print(10)
print(3.14)
print(10.)
print(.001)
print(1e100)
print(-3.14e-10)
print(0e0)
```

```
Examples of valid python number literals:
10
3.14
10.0
0.001
1e+100
-3.14e-10
0.0
```

2.2.1 Basic math operations with numbers

[44]:
```
# addition
print('1 + 1 =', 1+1)
```

```
1 + 1 = 2
```

[45]:
```
# with mixed number types
print('1.0 + 1 =', 1.0+1)
```

```
1.0 + 1 = 2.0
```

[46]:
```
# subtraction
print('17 - 25.2 =', 17-25.2)
```

```
17 - 25.2 = -8.2
```

[47]:
```
# multiplication
print('1234 * 0.789 =', 1234 * 0.78)
```

```
1234 * 0.789 = 962.52
```

[48]:
```
# be aware of the limited precision of floating point operations!
print('1234 * 0.789 =', 1234 * 0.789)
```

```
1234 * 0.789 = 973.6260000000001
```

[49]:
```
# division
print('1212.5 / 25 =', 1212.5 / 25)
```

```
1212.5 / 25 = 48.5
```

[50]:
```
# modulo, i.e. remainder
print('9 % 2 =', 9 % 2)
```

```
9 % 2 = 1
```

[51]:
```
# sign change
a = 9
print('-a =', -a)
```

```
-a = -9
```

[52]:
```
# absolute value
b = -15
print('abs(b) =', abs(b))
c = 23
print('abs(c) =', abs(c))
```

```
abs(b) = 15
abs(c) = 23
```

[53]:
```
# power operations
print('2 ** 3 =', 2**3)
```

```
2 ** 3 = 8
```

[54]:
```
# square root
print('16 ** 0.5 =', 16**0.5)
```

```
16 ** 0.5 = 4.0
```

[55]:
```
# using variables for calculations
x = 6
y = 4
x + y
```

[55]:
```
10
```

[56]:
```
'a' + 4
```

```
␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-56-833c9ffc6e69> in <module>
   ----> 1 'a' + 4


       TypeError: can only concatenate str (not "int") to str
```

[57]:
```
# python minds operator precence
print('2 + 2 * 2 =', 2 + 2 * 2)
print('... and not 8')
```

```
2 + 2 * 2 = 6
... and not 8
```

[58]:
```
# otherwise, use brackets if necessary
(2+2)*2
```

[58]:
```
8
```

[59]:
```
# assume, you compute ...
18-7
```

[59]:
```
11
```

[60]:
```
# ... but didn't save it to a variable.
# Still, you can use the last return value of python to do follow-up␣
 ↪calculations:
_+2
```

[60]:
```
13
```

[61]:
```
# how to round a number
round(1.9)
```

[61]:
```
2
```

[62]:
```
round(1.4987643)
```

[62]:
```
1
```

[63]:
```
round(1.4987643, 0)
```

[63]:
```
1.0
```

[64]:
```
# round to a specific digit
round(1.4987643, 5)
```

[64]:
```
1.49876
```

[65]:
```
round(1.4987643, 0)
```

[65]:
```
1.0
```

2.3 1.3 Typecasting

Typecasting is the change of a type of variable into another:

[66]:
```
# convert variable to a string
a = 5
b = str(a)

print('a is of type', type(a))
print('b is of type', type(b))
```

```
a is of type <class 'int'>
b is of type <class 'str'>
```

[67]:
```
# convert variable to an integer
a = '5'
b = int(a)

print('a is of type', type(a))
print('b is of type', type(b))
```

```
a is of type <class 'str'>
b is of type <class 'int'>
```

[68]:
```
# convert variable to a float
a = 5
b = float(a)
```

```
print('a is of type', type(a))
print('b is of type', type(b))
```

```
a is of type <class 'int'>
b is of type <class 'float'>
```

2.4 1.4 Polymorphism

Polymorphism is the **reuse** of operations, methods, attributes or function that can be defined differently depending on the variable type.

[69]:
```
"2" + "2" # plus operator can be used in strings, too
```

[69]:
```
'22'
```

[70]:
```
"Monty" + " " + "Python" # the plus operator simply glues the strings together
```

[70]:
```
'Monty Python'
```

[71]:
```
# not all operators all polymorph
"2" / "2"
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-71-16fcfd1e609f> in <module>
  1 # not all operators all polymorph
----> 2 "2" / "2"


TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

[72]:
```
"2" * 2 # some mixed variable type operations are also available
'22'
```

[72]:
```
'22'
```

[73]:
```
"A0" * 5 # a string multiplied with an integer strings together the string by␣
↪the number of the integer
```

[73]:
```
'A0A0A0A0A0'
```

[74]:
```
3 * 'un' + 'ium' # operators can be also combined
```

[74]:
```
'ununununium'
```

3 1.5 Naming convention for variables

Valid variable names start with a letter or an underscore **"_"**. Variable names are case sensitive, i.e. capital or lower letters are distinguished. Therefore, two variables with the name “House” and “house” can be used although it does not follow the naming convention of python.

The naming convention for variables is written in small letters. Variables containing more than one word are connected by underscores, e.g. maximum_length. Camel casing by writing the next word in a capital letter, e.g. MaximumLength, is not well accepted in the python community.

Additionally, the letter l and o should not be used as it can be mistaken by the numbers 1 and 0. More information can be found https://www.python.org/dev/peps/pep-0008/#prescriptive-naming-conventions