DataScientest • com

Python for Data Science

Introduction: Variables and Types

Introduction

> Python is a programming language with a readable, efficient and easy to learn syntax. It has become the most used language for Data Science thanks to many libraries such as:
>
> > - Numpy (http://www.numpy.org/) for scientific computing.
> > - Pandas (http://pandas.pydata.org/) for data processing and analysis.
> > - Matplotlib (https://matplotlib.org/) for plotting data.
> > - Scikit-learn (http://scikit-learn.org/stable/) to train machine learning models.
> > - And many more.
>
> Python is not limited to these packages and has other advantages which are essential to business integration, thus combining very well with web services and databases.

1. Variables

In [1]:

```
# Insert your code here
my_variable = 123456789

my_variable_times_2 = my_variable * 2

print(my_variable, my_variable_times_2)
```

```
123456789 246913578
```

Hide solution

In [ ]:

```
# Creation of the variables
my_variable = 123456789
my_variable_times_2 = my_variable * 2

# Display of the variables
print(my_variable)
print(my_variable_times_2)
```

Variables can have simple names such as `x` or `y`, but also more descriptive names like `transactions_customers_2018_2019` or `volume_total_sales_2020`. To name variables, there are several **rules** to follow:

- The name of a variable must **start** with a **letter** or an **underscore** (_).
- The name of a variable **cannot start** with a **number**.
- The name of a variable can only contain **alpha-numeric characters and underscores**.
- Variable names are **case sensitive**. `a_variable`, `A_variable` and `A_VARIABLE` are three **different** variables.

2. Lists: Indexation

A list is a very particular type of variable because it can contain **several** values and its creation is done with a specific syntax:

```
a_list = [2, 3.0, "Hello"]
```

> - This list contains **3 elements**: `2` , `3.0` and `"Hello"` .
> - The square brackets `[` and `]` determine the start and the end of the list.
> - The elements of the list are separated by **commas**.

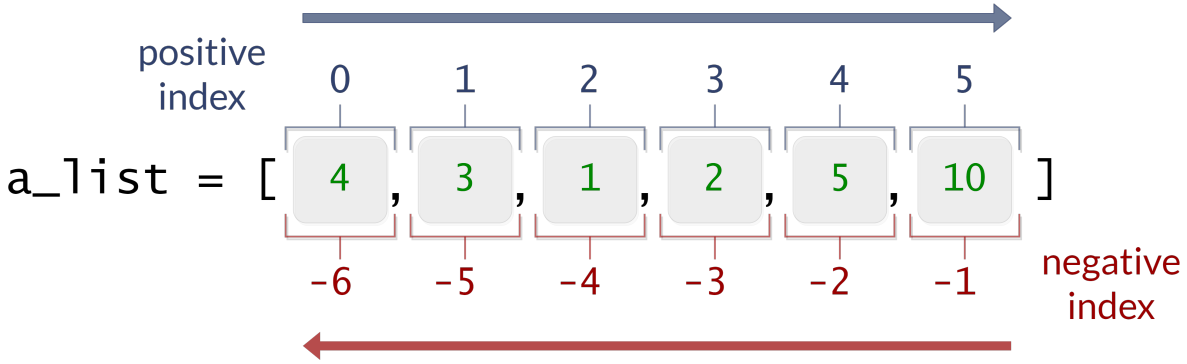
To access an element in the list, you must specify **its index** between square brackets:

```
# Display the first element of the list
print(a_list[0])
>>> 2
```

It is also possible to update elements of a list thanks to their index:

```
# Modification of the 2nd element of the list
a_list[1] = 4
```

As you may have noticed, the indexation of a list **starts at index 0**. It is also possible to index the elements in reverse order, this is called **negative indexing**.



Thus, we can retrieve the **last** element of a list with the index **`-1`** , the second to last element with the index `-2` and so on. This is very useful when we have very long lists with an unknown number of elements.

In [2]:

```
my_list = [4, -1, 2, -3, 3]

# Insert your code here
my_list[0] = -3
my_list[1] = -1
my_list[2] = 2
my_list[3] = 3
my_list[4] = 4

print(my_list)
```

```
[-3, -1, 2, 3, 4]
```

Hide solution

In [ ]:

```
my_list = [4, -1, 2, -3, 3]

# Update the elements of the list
my_list[0] = -3
my_list[1] = -1
my_list[2] = 2
my_list[3] = 3
my_list[4] = 4

# Display the list
print(my_list)
```

3. Lists: Slicing

Slicing is a special kind of indexation that is used to build new lists out of existing ones.

The use of slicing makes it possible to retrieve a **sub-list** of elements from a larger list by specifying the start and end indices of the sub-list.

```
my_list = [1, 5, "Hello", -1.4, "how", 103, "are", "you"]

# Retrieve the FIRST 4 elements of my_list
first_elements = my_list[0:4]

# Display these elements
print(first_elements)
>>> [1, 5, "Hello", -1.4]
```

THE ELEMENT AT THE END OF THE SLICING INDEX IS NOT INCLUDED IN THE SUB-LIST. Indeed, `my_list[0:4]` contains only the elements with indices `0`, `1`, `2` and `3`.

If the start index is not specified, then the slice will contain all the elements from the start of the list to the end index.

```
# Retrieve the FIRST 4 elements of my_list
first_elements = my_list[: 4]
```

Likewise, if the end index is not specified, then the slice will contain all the elements from the start index to the end of the list.

```
# Retrieve the LAST 3 elements of my_list
last_elements = my_list[-3:]
```

In [3]:

```
a_long_list = [-16, 6, -4, -18, 18, 20, 21, -6, 19, 25, 11,
               2, 9, 7, -16, 16, 4, -15, 11, 7, 17, 18, 4,
               25, 17, 28, -6, 17, 1, 14, -20, -15, 20, -15,
               -8, 8, -19, -11, -20, -16, 3, 3, -10, -5, 10,
               24, -1, 1, -10, 6, 10, -6, -14, 25, 8, -11,
               -17, -9, 0, 21, 3, 14, 7, 10, 25, 24, -18, -11,
               2, 29, 17, -6, 6, -11, 2, -18, 20, -15, -11,
               15, -10, 8, -15, 25, -15, 10, 28, -12, 11, 14,
               27, -1, 10, -2, -15, -10, 19, 26, 3, 27]

# Insert your code here

# First 10 elements of list
print(a_long_list[:10])

# Last 10 elements of list
print(a_long_list[-10:])
```

```
[-16, 6, -4, -18, 18, 20, 21, -6, 19, 25]
[27, -1, 10, -2, -15, -10, 19, 26, 3, 27]
```

Hide solution

In [ ]:

```
a_long_list = [-16, 6, -4, -18, 18, 20, 21, -6, 19, 25, 11,
               2, 9, 7, -16, 16, 4, -15, 11, 7, 17, 18, 4,
               25, 17, 28, -6, 17, 1, 14, -20, -15, 20, -15,
               -8, 8, -19, -11, -20, -16, 3, 3, -10, -5, 10,
               24, -1, 1, -10, 6, 10, -6, -14, 25, 8, -11,
               -17, -9, 0, 21, 3, 14, 7, 10, 25, 24, -18, -11,
               2, 29, 17, -6, 6, -11, 2, -18, 20, -15, -11,
               15, -10, 8, -15, 25, -15, 10, 28, -12, 11, 14,
               27, -1, 10, -2, -15, -10, 19, 26, 3, 27]

# Display the 10 first elements
print(a_long_list[:10])

# Display the last 10 elements
print(a_long_list[-10:])
```

4. Lists: Methods

So far, we have seen how to update the items in a list. In the following, we will see how to add or delete items from a list.

For this, we will use the **`insert`** and **`pop`** **methods**.

In *Object Oriented Programming*, a **method** is a function specific to a **class** of objects (the list class in our case).

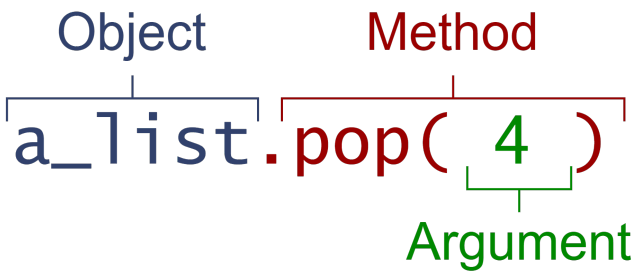
The **`pop`** method of the list class allows us to delete an element from a list at a specified index:

```
my_list = [1, 5, "Hello", -1.4, "how", 103, "are", "you"]

# Delete element at index 4
my_list.pop(4)

# Display the list
print(my_list)
>>> [1, 5, 'Hello', -1.4, 103, 'are', 'you']
```

The syntax for using an object's method is very specific:



- The object calling the method must already exist.
- The name of the method must be followed by **parentheses** containing the **arguments** of the method. These arguments will modify the method's execution.
- The name of the object and the method are separated by a dot **`.`**.

The `pop` method takes only a **single argument**, the index of the element to remove.

In [4]:

```
my_list = [1, 5, "Hello", -1.4, "how", 103, "are", "you"]

# Insert your code here


my_list.pop(2)
my_list.pop(3)
my_list.pop(4)
my_list.pop(-1)
my_list
```

Out[4]: [1, 5, -1.4, 103]

Hide solution

In [ ]:

```
my_list = [1, 5, "Hello", -1.4, "how", 103, "are", "you"]

# Deletion of "Hello" which is at the 3rd position (index 2)
my_list.pop(2)

# Deletion of "how" which is at the 4th position (index 3)
my_list.pop(3)

# Deletion of "are" which is at the second to last position
my_list.pop(-2)

# Deletion of "you" which is at the last position
my_list.pop(-1)


# Display of the list
print(my_list)
```

To add a new value to a list, we can use the **`insert`** in which **2 arguments** are passed:

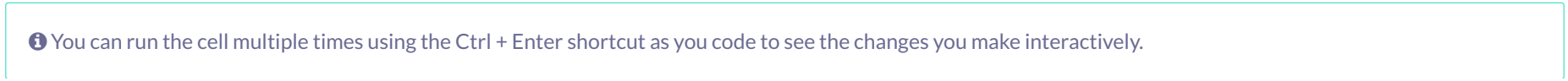
- The first argument is the **index** where we want to insert the value.
- The second argument is the **value** we want to insert.

```
# Insertion of the value "Hello" at index 2
my_list.insert(2, "Hello")
```

When a method takes several arguments, they must be separated by **commas**.

- **(c)** Remove **all numbers** from the `my_list` list using the `pop` method.
- **(d)** Insert in `my_list` the elements `"Bonjour"`, `"comment"`, `"ça"` and `"va"` at the appropriate indices so that printing `my_list` displays:

  `["Hello", "Bonjour", "how", "comment", "are", "ça", "you", "va"]`



In [5]:

```
my_list = [1, 5, "Hello", -1.4, "how", 103, "are", "you"]

# Insert your code here
my_list.pop(0)
my_list.pop(0)
my_list.pop(1)
my_list.pop(2)

my_list.insert(1, "Bonjour")
my_list.insert(3, "comment")
my_list.insert(5, "ça")
my_list.insert(7, "va")


# Display of the list
print(my_list)
```

```
['Hello', 'Bonjour', 'how', 'comment', 'are', 'ça', 'you', 'va']
```

Hide solution

In [ ]:

```
my_list = [1, 5, "Hello", -1.4, "how", 103, "are", "you"]

# Removal of numbers
my_list.pop(0)
my_list.pop(0)
my_list.pop(1)
my_list.pop(-3)

# Insertion of the elements "Bonjour", "comment", "ça" and "va"
my_list.insert(1, "Bonjour")
my_list.insert(3, "comment")
my_list.insert(5, "ça")
my_list.insert(7, "va")

# Display of the list
print(my_list)
```

> It is possible to add an element directly **at the end** of a list using the **append** method.
>
> ```
> # Addition of the element "Goodbye" at the end of the list
> my_list.append("Goodbye")
> ```
>
> This method is used very often when a list is gradually extended. For example, when we want to store the values taken by a changing variable over time.

- **(e)** Launch the following cell to create a variable `x` of value `0` and a list whose unique element will be `x`.

In [6]:

```
x = 0
a_list = [x]
```

> We can now modify `x` and store the new value of `x` in the list.

- **(f)** Run the following cell several times:
  - The value of `x` is incremented by 1.
  - The new value of `x` is added at the end of the list `a_list`.

In [9]:
```
x = x + 1
a_list.append(x)
print(a_list)
```

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

> To merge two lists, we can use the **`extend`** method. The argument of the `extend` method is a list of elements that we want to add to the end of the list **calling** the method.
>
> It is also possible to use the `+` operator, but it is not recommended as it can add ambiguity to the code when we are not sure whether we are working with numbers or lists.

In [10]:
```
### Merging 2 lists with the extend method

list_1 = ["Hello", "how", "are", "you", "?"]
list_2 = ["Fine", "and", "you", "?"]

# Merging the elements of list_2 with list_1
list_1.extend(list_2)

# Display of list_1
print(list_1)
```

```
['Hello', 'how', 'are', 'you', '?', 'Fine', 'and', 'you', '?']
```

In [11]:
```
### Merging 2 lists with the + operator

list_1 = ["Hello", "how", "are", "you", "?"]
list_2 = ["Fine", "and", "you", "?"]

# Merging the elements of list_2 with list_1
list_1 = list_1 + list_2
print(list_1)
```

```
['Hello', 'how', 'are', 'you', '?', 'Fine', 'and', 'you', '?']
```

The list class contains other methods as well. The methods we have seen so far are summarized in the table below:

Method	Argument	Description
pop	index	Removes the element from the list located at the specified index
insert	index, value	Adds a new element to the list at the specified index
append	value	Adds the value at the end of the list
extend	list	Merges the list calling the method with the list in argument

All object classes have methods to manipulate them. The notion of object classes and methods will be explored later in your training, but it is the **heart** of high-level programming with Python.

5. Tuples

Tuples are a data structure similar to lists:

```
# Create a tuple
a_tuple = ("Hello", -1, 133)
a_tuple = "Hello", -1, 133    # These syntaxes are equivalent

# Display the first element of the tuple
print(a_tuple[0])
>>> Hello

# Display the last element of the tuple
print(a_tuple[-1])
>>> 133
```

- The definition of a tuple is done **with or without parentheses**.
- The indexing of a tuple is **identical** to that of a list.

A very important characteristic of tuples is that **they are immutable**, i.e. they cannot be modified through indexing.

In [12]:

```
a_tuple = "Hello", -1, 133
print(a_tuple)

# Tuple assignment
x, y, z = a_tuple

print(x)
print(y)
print(z)
```

```
('Hello', -1, 133)
Hello
-1
133
```

The variables `x`, `y` and `z` were created and assigned with values simultaneously. For the tuple assignment to work correctly, there must be **as many** variables to assign as there are **elements in the tuple**.

Tuple assignment gives an elegant syntactic solution to the problem of **exchanging values**: We have two variables `a` and `b` and we want to exchange their values, that is to say that `a` must take the value of `b` and `b` the value of `a`.

In more classic programming languages, we would have to create a temporary **variable** which contains one of the values of `a` or `b`:

```
# We store the value of a in a temporary variable
tmp = a

# We overwrite a with the value of b
a = b

# We overwrite b with the value of the temporary variable
b = tmp
```

Thanks to the tuple assignment, this operation can be done in a single line of code:

```
# Exchange of values between a and b
a, b = b, a
```

In [13]:

```
a = 10
b = 5

a, b = b, a

print(a)
print(b)
```

```
5
10
```

6. Dictionaries

Lists and tuples are data structures whose elements are indexed by **integers** in an **orderly** fashion.

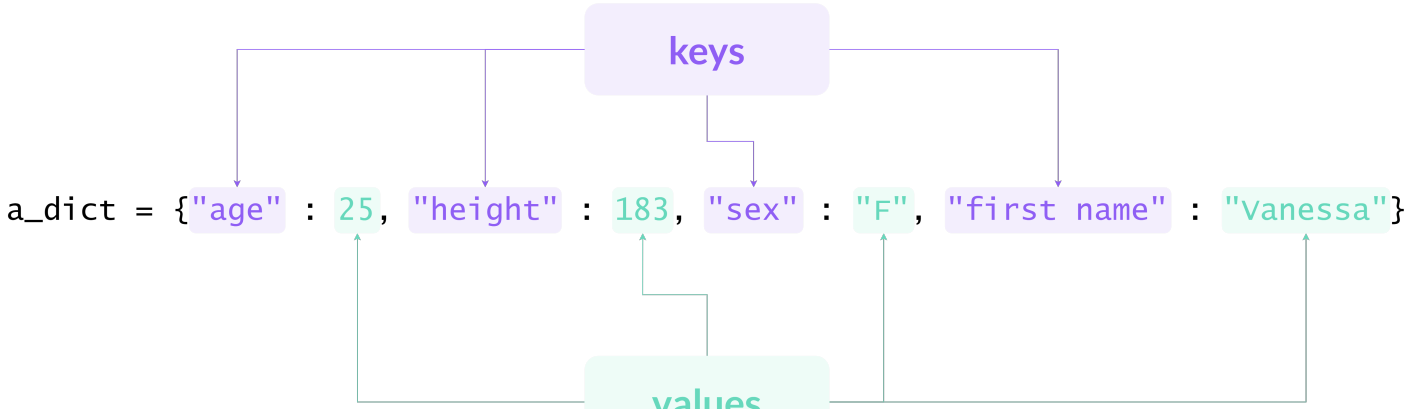
Dictionaries are a very special data structure because the elements of a dictionary can be **indexed** freely by **numbers**, **character strings** and even **tuples**.

Dictionaries are very useful for storing information:

```
# Definition of a dictionary
a_dict = {"age": 25,
          "height": 183,
          "sex":"F",
          "first name":"Vanessa"}

# Display of elements
print(a_dict["age"])
>>> 25

print(a_dict["first name"])
>>> Vanessa
```



In [14]:
```
# Insert your code here
# Create Dictionary Card_id

card_id = {"first name" : "Paul",
           "last name" : "Lefebvre",
           "emission" : 1978}

print(card_id)

# Change first name

card_id["first name"] = "Guillaume"

print(card_id)
```

```
{'first name': 'Paul', 'last name': 'Lefebvre', 'emission': 1978}
```

Out[14]:
```
{'first name': 'Guillaume', 'last name': 'Lefebvre', 'emission': 1978}
```

Hide solution

In [ ]:
```
# Definition of the dictionnary
card_id = {"first name": "Paul",
           "last name" : "Lefebvre",
           "emission"  : 1978}
print(card_id)

# Overwriting a field of the dictionnary
card_id["first name"] = "Guillaume"
print(card_id)
```

> It is possible to **add** new keys to our dictionary very easily by simply assigning a value to a new key:
>
> ```
> # Adding a new key to the dictionary
> a_dict["new key"] = a_value
> ```
>
> As for lists, it is possible to delete an element using the **pop** method. Instead of specifying the index to delete, you must enter the **key**:
>
> ```
> # Deletion of the key "a key"
> a_dict.pop("a key")
> ```

- **(c)** Add a new key **`"expiration"`** to the `card_id` dictionary by assigning it the value `1993` and display it.

In [16]:

```
# Insert your code here

# (c) Add a new value to dictionary
card_id["expiration"] = 1993

print(card_id)

# (d) How long was this ID card valid for?
print(f" This Id card is valid for {card_id['expiration'] - card_id['emission']} years.")
```

```
{'first name': 'Guillaume', 'last name': 'Lefebvre', 'emission': 1978, 'expiration': 1993}
 This Id card is valid for 15 years.
```

Hide solution

In [ ]:

```
# Adding a new key to the dictionnary of a new key
card_id["expiration"] = 1993
print(card_id)

# How long was the card valid for?
validity_duration = card_id["expiration"] - card_id["emission"]

print("This ID card was valid for", validity_duration, "years.")
```

Conclusion

Lists, tuples and dictionaries are indexable **variables** that can contain many different elements.

In this introductory notebook, we learned how to manipulate them and the syntax we used will be the same for **all** indexable objects that we will use later such as **databases**.

- Access to the elements of an indexable variable is done via square brackets **[ ]** by entering:

  - For lists and tuples the **positional index** of a value. The indices of a list or a tuple always start from **0**. It is also possible to access several indices by **slicing**.
  - For dictionaries: the **key**.

Each indexable type has its specific symbol for its creation:

- For a **list**, we use **square brackets**: **[ ]**
- For a **tuple**, the **parentheses**: **( )** (or nothing)

✖ Unvalidate