02.quick_python_overview

November 11, 2022

0.0.1 1. Variables in Python

[1]:
```
my_string = 'Welcome to the Genetics and Genomics class!'
print(my_string)
```

```
Welcome to the Genetics and Genomics class!
```

0.0.2 2. Data types and operations

Python uses classes to define data types. The most frequently used data types are: `str`, `int`, `float`, `range()`, `list()`, `set()`, `dict()`, `bool()`, `None` (and there are some more).

2.1. Numeric types in Python There are three numerical types in Python, which include integers (`int`), floating point numbers (real values, aka `float`) and complex numbers (`complex`). Here are some examples:

[2]:
```
int_number = 3
float_number = 1.5

print('Integer number is ', int_number, 'and its type is', type(int_number))
print('Floating number is ', float_number, 'and its type is',␣
 ↪type(float_number))
```

```
Integer number is  3 and its type is <class 'int'>
Floating number is  1.5 and its type is <class 'float'>
```

You can perform standard arithmetical operations numerical types, including: * Addition + * Subtraction - * Multiplication Division / * Exponentiation ** * Floor division // * Modulus %

[3]:
```
x = 2
y = 3
print(x / y, round(x / y, 3))  # round() function allows rounding the float to␣
 ↪n decimals, here n=3
```

```
0.6666666666666666 0.667
```

Important: to check arguments of functions (e.g. the function named `my_function()`), use one of the options below: * `?my_function` * `help(my_function)` * Put the pointer inside the function and press `shift + tab`

[4]:
```
# For example:
help(round)
```

```
Help on built-in function round in module builtins:

round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None.  Otherwise
    the return value has the same type as the number.  ndigits may be negative.
```

2.2 Booleans Boolean values are either `True` or `False`. Note that integers `0` and `1` are equivalent to `False` and `True` respectively.

For instance, we can compare the numbers with the operators less (`<`), greater (`>`), equal (`==`), not equal (`!=`) and, of course, less or equal (`<=`) and greater or equal (`>=`). This will yield a boolean output:

[5]:
```
print('The statement 15 < 1 is', 15 <= 1)
print('The statement 10 != 12 is', 10 != 12)
print('-' * 10)
print('The statement 0 == False is', 0 == False)
print('The statement 1 == True is', 1 == True)
```

```
The statement 15 < 1 is False
The statement 10 != 12 is True
----------
The statement 0 == False is True
The statement 1 == True is True
```

2.2.1 Logical operators `and`, `or`, `not` The logical operators `and`, `or`, `not` can be used to test different statements to get a boolean answer, for example:

[6]:
```
print((15 < 1) and (150 < 10))
print((15 > 1) and (150 < 10))

print((15 > 1) or (150 < 10))
print(not ((15 > 1) or (150 < 10)))
```

```
False
False
True
False
```

2.1. Strings In the previous section you have already used the variables of the type string (`str` in Python). Let's see what we can do with them!

[7]:
```
my_name = 'Lola'
```

One can use built-in methods to change the string, i.e. replace a substring (`.replace()`), make all letters lowercase (`.lower()`) or uppercase (`.upper()`), for example:

[8]:
```
print(my_name.replace('la', 'ic'))
print(my_name.lower())
print(my_name.upper())
```

```
Loic
lola
LOLA
```

[9]:
```
# Let's save the transformed name into a new variable
new_name = my_name.replace('la', 'ic')
```

You can concatenate strings by using the `+` operator as follows:

[10]:
```
print(my_name + ' and ' + new_name + ' are friends')
# Note that concatenation happens without spaces!
```

```
Lola and Loic are friends
```

To get a string that would be joined with the same separator (it can be comma, semicolon, dot, underscore, etc.), use the `.join()` method. To split a string, use `.split()` function, which allows splitting by a defined pattern:

[11]:
```
long_string = '_'.join([my_name, new_name, 'Robert', 'and', 'Sarah', 'are',␣
↪'friends'])
print(long_string)
print(long_string.split('_'))
```

```
Lola_Loic_Robert_and_Sarah_are_friends
['Lola', 'Loic', 'Robert', 'and', 'Sarah', 'are', 'friends']
```

Strings in Python are represented as arrays of Unicode characters and therefore, single elements of a string can be accessed through the square brackets. To acces several elements of the string, use colon and specify the index range in the `[start_idx:end_idx]` format as shown below. Note that it is not necessary to specify start or end of the slice if start/end match the sequence start/end coordinate.

[12]:
```
print(my_name[0], my_name[3])
print('The first half of the name is:', my_name[:2])
```

```
L a
The first half of the name is: Lo
```

! Important: *Enumeration in Python starts from **0** ! That is why the first index of the "L" letter in the variable `my_name` is accessed with `[0]`. When slicing a string keep in mind that the last coordinate is **not included**. One can access elements from the end of the string/list by using negative indexers starting from -1. In this case, the last element of `my_name` will be accessed as `my_name[-1]`. To acces a slice from the end, use the following syntax:*

2.2. Lists Lists in Python are created with square brackets and may contain mixed data types, multiple occurrences of the same element. For example:

[13]:
```
my_list = ['flowers', 20., 'cat', 'dog', 2, True, 'flowers', ['a', 'b', 5]]
```

Elements in the list are indexed, so to access a single element of the list or a several elements, use the following options:

[14]:
```
print('The first element of my_list is', my_list[0])
print('The last two elements of my_list are', my_list[-2:])
print('Reversed list is:', my_list[::-1])
```

```
The first element of my_list is flowers
The last two elements of my_list are ['flowers', ['a', 'b', 5]]
Reversed list is: [['a', 'b', 5], 'flowers', True, 2, 'dog', 'cat', 20.0,
'flowers']
```

[15]:
```
my_list.append(my_name)
print(my_list)
print('New length =', len(my_list))
```

```
['flowers', 20.0, 'cat', 'dog', 2, True, 'flowers', ['a', 'b', 5], 'Lola']
New length = 9
```

Note that `my_list` contains a sublist. Its elements can be accessed by first extracting the list based on its index in `my_list`, and then extracting an element inside the sublist by its index starting from `0` and to the length of the sublist:

[16]:
```
print(my_list[7])  # get the sublist
print(my_list[7][0])  # extract the first element of the sublist
print(my_list[7][:2])  # extract the fisrt two elements of the sublist
```

```
['a', 'b', 5]
a
['a', 'b']
```

Side note 1: *Note that there are two methods for sorting list: `my_list.sort()` orders elements inplace (changes the initial list), and `sorted(my_list)` returns a sorted copy of the list.*

2.3. Sets Sets in Python represent a non-repetitive collection of elements, so it is the set in a mathematical sense. Therefore, sets allow various set-like operations including built-in methods, such as `.intersection()`, `.union()`, `.difference()`. Similarly to lists, you can add/remove

elements to/from a set, but sets cannot be sorted, set elements cannot be changed because sets are not indexed.

Side note: *To create an unchangeable set, use the* `frozenset()` *method. To read more about it, check out the documentation.*

You can create a set with curly braces by listing the elements inside:

[17]:
```
my_set = {'flowers', 20.0, 'pigeon', 'parrot', 2, True, 'flowers'}
print(my_set)
```

```
{'flowers', True, 2, 'pigeon', 20.0, 'parrot'}
```

Even though the element 'flowers' is repeated twice, the set definition with `{}` will automatically remove this repetition.

Alternatively, sets can be created from lists, for example, by using the `set()` constructor.

Below you can find several examples of set operations, such as intersection, union, difference:

[18]:
```
my_list = ['flowers', 20.0, 'cat', 'dog', 2, True, 'flowers']
```

[19]:
```
my_set = set(my_list)
new_set = {'a', 'b', 'c', 20, 'flowers'}

print('Intersection of sets:', my_set.intersection(new_set))
print('Union of sets:', my_set.union(new_set))
print('Difference of sets, option 1:', my_set.difference(new_set))

# Equivalently, the set difference can be obtained as follows:
print('Difference of sets, option 2:', my_set - new_set)

print('Symmetric difference of sets:', my_set.symmetric_difference(new_set))
# Symmetric difference of sets can be obtained as (my_set - new_set).
 ↪union(new_set - my_set)
```

```
Intersection of sets: {'flowers', 20}
Union of sets: {'flowers', True, 2, 'dog', 'b', 'c', 20.0, 'cat', 'a'}
Difference of sets, option 1: {True, 2, 'cat', 'dog'}
Difference of sets, option 2: {True, 2, 'cat', 'dog'}
Symmetric difference of sets: {True, 2, 'dog', 'b', 'c', 'cat', 'a'}
```

To know more about set methods in Python, check out this link.

2.4. Dictionaries The last, but not less useful data type is dictionary. Dictionaries are powerful since they allow to store different data types in the `{key: value}` format, which allows efficient hashing of elements. Dictionaries represent ordered (since Python3.6) structures with no repetitive keys, but editable elements. Note that keys can be represented only by immutable data types, check out this link to know more about mutable and immutable data types.

Dictionaries are represented with curly brackets and the following syntax:

[20]:
```
my_dict = {
    'name': my_name,
    'age': 20,
    'favourite_animal': 'dog',
    'sibling_names': ['John', 'Sarah']
}
print(my_dict)
```

```
{'name': 'Lola', 'age': 20, 'favourite_animal': 'dog', 'sibling_names': ['John',
'Sarah']}
```

[21]:
```
print(my_dict.items())  # get all dictionary items (keys and values)
print(my_dict.keys())  # get only dictionary keys
print(my_dict.values())  #get only dictionary values
```

```
dict_items([('name', 'Lola'), ('age', 20), ('favourite_animal', 'dog'),
('sibling_names', ['John', 'Sarah'])])
dict_keys(['name', 'age', 'favourite_animal', 'sibling_names'])
dict_values(['Lola', 20, 'dog', ['John', 'Sarah']])
```

To get values for a particular key of the dictionary, use the square brackets as follows:

[22]:
```
my_dict['age'], my_dict.get('age')
```

[22]:
```
(20, 20)
```

[23]:
```
my_dict.get('eye_color'), my_dict.get('eye_color', 'no info')
```

[23]:
```
(None, 'no info')
```

To get more information on dictionary methods, check out this link.

0.0.3 3. Conditional statements (if, elif, else)

To test several statements, you can use the `elif` statement, which will tell the interpreter that when the `if` statement is not satisfied, try another contition, and if it works – great! Otherwise, proceed either to the `else` statement, or execute the following lines of code (if any). Note that conditional statements in Python can be nested.

[24]:
```
x, y, z = 1, 3, 5
if (x + y) == z:
    # execute some code here
    print('The sum of x and y is equal to z')
elif (x + y) < z:
    # execute some code here
    print('The sum of x and y is less than z')
else:
    # execute some code here
```

```
    print('None of the conditions is satisfied')
```

```
The sum of x and y is less than z
```

0.0.4 4. While and for loops

4.1. Writing loops with `while` `While` is a conditional looping statement that allows you to perform an operation (or many operations) until the `while` statement is satisfied. Let's try solving the problem of finding all numerical values in `my_list` and storing them to a new list.

[25]:
```
print(my_list)
```

```
['flowers', 20.0, 'cat', 'dog', 2, True, 'flowers']
```

[26]:
```
counter = 0  # Since enumeration in Python starts from 0, we set counter to 0␣
↪and change it in the loop
numerical_list = []  # Create an empty list
while counter < len(my_list):
    list_element = my_list[counter]  # get the element of my_list with the␣
↪counter as index
    if type(list_element) == int or type(list_element) == float:  # Check data␣
↪type of the element
        numerical_list.append(list_element)  # Append a numerical value to the␣
↪new list
    # Increase the counter by one to access next element of the my_list
    counter += 1  # This is equivalent to: counter = counter + 1
```

[27]:
```
print(numerical_list)
```

```
[20.0, 2]
```

4.2. Writing loops with `for` The `for` loop is useful when you need to iterate over a sequence of elements, which can be represented as a string, set, list, tuple or dictionary. Loops can be combined with conditional statements and various python functionalities. Please see the subsections below for more examples.

4.2.1. Iterating over lists As in the subsection above, we will use `my_list` to find numerical data types.

[28]:
```
print(my_list)
```

```
['flowers', 20.0, 'cat', 'dog', 2, True, 'flowers']
```

[29]:
```
numerical_list = []  # create an empty list
for list_element in my_list:  # iterate over list elements
```

```
    if type(list_element) == int or type(list_element) == float:  # check␣
↪element type
        numerical_list.append(list_element)  # append numerical value to list
```

[30]:
```
print(numerical_list)
# It worked!
# We have added a numerical element from a sublist to all other numerical␣
↪values!
```

```
[20.0, 2]
```

4.2.4. Iterating over dictionaries

[31]:
```
for key in my_dict.keys():  # iterate over keys
    print(key, my_dict[key])  # print dictionary key and the respective value
```

```
name Lola
age 20
favourite_animal dog
sibling_names ['John', 'Sarah']
```

[32]:
```
for value in my_dict.values():  # iterate over values
    print(value)  # print dictionary value
```

```
Lola
20
dog
['John', 'Sarah']
```

[33]:
```
for key, value in my_dict.items():  # iterate over both keys and values
    print(key, value)  # print dictionary key and value
```

```
name Lola
age 20
favourite_animal dog
sibling_names ['John', 'Sarah']
```

One of the useful functions is `zip()`, which takes two or more iterable objects (e.g., lists) and joins them into `tuple`. Let's see how we can create a dictionary from two lists with the `zip()` function:

[34]:
```
list_with_keys = ['name', 'eye_color', 'favourite_animal']
list_with_values = ['Lola', 'green', 'dog']
new_dict = {}
for key, value in zip(list_with_keys, list_with_values):
    new_dict[key] = value
```

[35]:
```
# To view the content, first convert the zip object into list
print(list(zip(list_with_keys, list_with_values)))
```

```
[('name', 'Lola'), ('eye_color', 'green'), ('favourite_animal', 'dog')]
```

0.0.5 5. Creating custom functions

[36]:
```python
def sum_values_in_list(input_list):  # define function name and its arguments
    sum_ = 0
    for element in input_list:
        sum_ += element
    return sum_  # the output to return
```

Let's try out these functions! But first, we need to create some input data:

[37]:
```python
lst_1 = [i for i in range(5)]
lst_2 = [m / 10 for m in range(10)]
```

[38]:
```python
print('Sum of values in the first list =', sum_values_in_list(lst_1))
print('Sum of values in the second list =', sum_values_in_list(lst_2))
```

```
Sum of values in the first list = 10
Sum of values in the second list = 4.5
```

Great, let's continue to the next notebook!