2 Introduction to the syntax of Python. Part II

2.1 Lists

From a very general standpoint *programming* is about *algorithms* and *data structures*. First, and most basic for us in this course, data structure in Python is `list`, which we already saw in the previous section. Recall that a list is Python is a sequence of objects (any objects), divided by commas, and bounded by square parenthesis: e.g., `my_list = [1, 0, 1, 'Artem', [5, "blue"]]`. The elements of the list can be accessed by using index, which starts from 0, so that `my_list[0] = 1, my_list[4] = [5, "blue"]`. Built-in function `len` returns the length of a list: `len(my_list) = 5`.

Before discussing various ways dealing with lists in Python, let me emphasize a very important distinction of lists from, say, `int` or `float` type data. Consider the following code.

```
a = 5
b = a      #b is assigned value 5

a = 10     #I change the value of a

#What is b now?
>>> print(b)
5          #b is still 5 and a is 10 as expected
```

Now let me try to do something similar with lists.

```
list1 = [1, 2, 3]
list2 = list1     #here, if you access list2, you will get [1, 2, 3]

list1[0] = 10     #I change the first object in list1

>>>print(list1)
[10, 2, 3]

#What will happen now?
>>>print(list2)
[10, 2, 3]

list2[1] = 'end'  #now list2 = [10, 'end', 3]
>>>print(list1)
[10, 'end', 3]
```

The explanation of this behavior is beyond my course (you can try to read about *mutable* and *not mutable* objects in Python), but this has to be kept in mind that when I do something like `list2 = list1` I do not create a copy of my list, all I am creating is a different name that points to exactly the same object, which now can be changed by using different names (if you have experience programming in C, recall *pointers*). Opposite to it, when I assign `b = a` I create a completely new object with name `b` and value 5. Now I can change `a`, this will not change `b`.

Math 488/688: Numerical Analysis
e-mail: artem.novozhilov@ndsu.edu. Fall 2021

So, how to make a copy instead of making just another name for a list? For instance, `list2 = list(list1)`.

Somewhat similar to the function `range` (recall the previous section), I can use so-called *slicing* to access multiple elements of a list: `list1[start:stop:step]`, note that `stop` is not included, the same as in `range`. I can also use negative indices, to access elements of a list from the end: `list1 = [1, 2, 3], list1[-1] = 3`.

Here are some examples of working with lists.

```
>>>list1 = ["blue", "green", "yellow", "grey", "orange", "red", "black", "white"]


>>>list1[:]   #all elements
["blue", "green", "yellow", "grey", "orange", "red", "black", "white"]

>>>list1[:4]  #all elements up to (but not including) the fifth element
['blue', 'green', 'yellow', 'grey']

>>>list1[1:4] #elements with indexes 1, 2, 3
['green', 'yellow', 'grey']

>>>list1[0:-1:2]#from beginning to end with step 2
['blue', 'yellow', 'orange', 'black']

>>>list1.append("pink") #add a new item to the existing list
>>>list1
['blue', 'green', 'yellow', 'grey', 'orange', 'red', 'black', 'white', 'pink']

>>>list2 = ["dark", "bright"]
>>>list1 + list2 #list concatenation
['blue', 'green', 'yellow', 'grey', 'orange', 'red', 'black', 'white', 'pink', 'dark',
    'bright']

>>>[0]*10 #an easy way to create a list of necessary length
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Now consider the problem to reverse a given list. A possible solution is given below, please make sure that you understand what is done in each line of the code.

```
#reverse a list

print(list1)

n = len(list1)

for i in range(n//2):
    list1[i], list1[n-1-i] = list1[n-1-i], list1[i]

print(list1)

#the result is
```

```
['blue', 'green', 'yellow', 'grey', 'orange', 'red', 'black', 'white', 'pink']
['pink', 'white', 'black', 'red', 'orange', 'grey', 'yellow', 'green', 'blue']
```

We could have done it much easier: `list1.reverse()`, which does exactly the same, but the message here is that any build-in method is not just a magical command, which instantaneously does what you need; on the opposite, behind these commands there is usually an algorithm that requires to perform a certain number (sometimes quite large) of operations, so do not overuse these existing commands.

Finally, one important thing in Python is the so-called *list comprehension*, which allows creating new lists in a very concise form.

Consider a problem of making a list of `n` integers squared. A possible straightforward solution is below.

```
n = 10

squares = [] #initially the list is empty

for i in range(1, n + 1):
    squares.append(i*i)

print(squares)

#the result is
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Instead I can write the following.

```
n = 10
squares = [x*x for x in range(1, n + 1)]
```

which will produce the same result. Note the syntax, I perform computing of some expression for the variable in the given range and produce a required list.

Assume now that I need to produce a list of squares of all even numbers from 1 to `n`. Here is a solution.

```
n = 10
even_squares = [x*x for x in range(1, n + 1) if x % 2 == 0]
print(even_squares)

#the result is
[4, 16, 36, 64, 100]
```

Not only list comprehension allows very concise code, it is actually much more efficient compared with the solution that is based on appending each time an element to a given list. You should definitely use this technique in Python.

I will use lists to represent *vectors* and *matrices*, so in the latter case I will be dealing with a list of lists. Here is a simple example: I create two four by five (note the order and compare with the code below) matrices and add them together, the result is a new matrix, which I also initially allocate with

all zeros.

```
A = [[1]*5 for i in range(4)]
B = [[2]*5 for i in range(4)]
C = [[0]*5 for i in range(4)]

for i in range(len(A)):
    for j in range(len(A[0])):
        C[i][j] = A[i][j] + B[i][j]

print(C)

#the result is
[[3, 3, 3, 3, 3], [3, 3, 3, 3, 3], [3, 3, 3, 3, 3], [3, 3, 3, 3, 3]]
```

2.2 Functions

Functions in programming are much more than functions in mathematics. Not only they allow to receive some parameters as an input and produce some result as an output similar to the mathematical functions, they are also used to divide the code into manageable, reusable, and easy to test parts, such that the whole program can be built out of existing (and error free) functions.

We already saw a number of *built-in* functions, such as `print`, `input`, etc. In addition to it, I can define my own functions in the following way.

```
def my_function_name(param1, param2):
    #note the indentation!
    body_of_function
    return result #this is an optional line
```

That is, I use the key word `def` to start the definition, then I use a name for my function. In the round parenthesis I list the parameters for my function, which will be used in the calculations, next there should be some code that performs certain required operations, and finally I return some results (which is optional). Note that any function can contain multiple `return`. Meeting this command halts the function operation.

Here is an example. Recall that $n!$, reads "n factorial" is defined as $n! = 2 \cdot \ldots \cdot n, 0! = 1, 1! = 1$. Let me write a Python function that computes it.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1

    result = 2

    for i in range(3,n+1):
        result = result * i #can also write result *= i
    return result
```

After I defined this function, I can use it by simply calling `factorial(10)` which will return the value

3628800.

Similar to what I discussed about lists above, you should be careful when passing lists to the functions. Here is a self-explanatory example.

```
def f_int(x):
    x = x + 1
    return x

>>>a = 5
>>>f_int(a)
6
>>>a
5

#for lists it will be different
def f_list(x):
    x[0] = 'True'
    return x

>>> list1 = ['False']*5
['False', 'False', 'False', 'False', 'False']

>>>f_list(list1)
['True', 'False', 'False', 'False', 'False']
>>>list1
['True', 'False', 'False', 'False', 'False']
```

There is much more to the Python functions, and I will be introducing some of convenient additional syntax you can use as it will be required.

2.3 Packages

Since we are doing mathematics, we need all the library of standard mathematical functions, like sine, cosine, exponent, etc. But typing something like `sin(3.14)` will return an error. To be able to use these functions we will need to request a *package* that contains these functions to be uploaded.

```
import math as ma #this line uploads package math into
                  #the system and also makes abbreviation for it

help(ma) #will show you a list of functions you can now use.

#now we can use it, e.g.
>>>ma.sin(ma.pi)
1.2246467991473532e-16 #note that the result is not zero!
```

If you just type `import math` you will have to use function names of the form `math.sin`, `mathy.pi`, etc.

I will be introducing different packages as I will need them.

2.4 Examples

Here I collect a few examples of programming some relatively simple and short Python functions. I try to use some additional features, which I explain below. Your task is to work through each line of the code to make sure you understand what is happening.

Example 2.1. Write a function that accepts two matrices and, if possible, multiplies them. The output is the product. Recall that matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ can be multiplied only if the number of columns of the first matrix is equal to the number of rows of the second one. So, if $\boldsymbol{A}$ is an $m \times k$ matrix and $\boldsymbol{B}$ is an $k \times n$ matrix then the result will be a $m \times n$ matrix $\boldsymbol{C} = \boldsymbol{AB}$ with the elements

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}.$$

Here is a function that does what is required.

```
def prod(A: list, B: list) -> list:
    '''takes two matrices A and B and
    produces, if possible, its product,
    which is returned as a result.
    '''

    rowsA, colA, rowsB, colB = len(A), len(A[0]), len(B), len(B[0])

    if colA != rowsB:
        return print("Matrices cannot be multiplied")

    #allocate the result of the product
    C = [[0]*colB for i in range(rowsA)]

    for i in range(rowsA):
        for j in range(colB):
            for l in range(colA):
                C[i][j] +=A[i][l]*B[l][j]

    return C
```

First note a new commentary at the beginning of the function, which is enclosed by triple quotes. This is a description, which the user will see if they type `help(prod)` in the notebook. In the definition of my function I indicate that I expect my arguments to be lists (e.g., `A: list`) and the output is also list: `-> list`. This is completely optional, does not perform any type checking and is given only for future readers of the code, it is up to you whether to use this feature of Python.

I assume that the user is careful enough to provide my function with matrices, i.e., lists of lists, but I check inside whether these two matrices can be multiplied (to be able to multiply matrices the number of rows of the first one should be equal to the number of rows of the second one). After it I just program the formula from above, creating initially a template for the product, which is $m \times n$ matrix of zeros. Inside the most inner loop I use the syntax `i += 1` which is equivalent to `i = i + 1`. Similarly, you can use `-=, *=, /=, //=, %=`.

To test this function I will create two matrices, filling them with random 0 and 1. For this purpose I upload package `random` and use its function `random.randint(a,b)`, which produces a random integer from the interval $[a, b]$, where a, b are integers.

```
import random

m = 3 #number of rows of the first matrix
k = 4 #number of columns of the first matrix and number of rows of the second
n = 2 #number of columns of the second matrix

#generate a random matrix m by k
A = [[random.randint(0,1) for i in range(k)] for j in range(m)]

#generate a random matrix k by n
B = [[random.randint(0,1) for i in range(n)] for j in range(k)]

# A = [[1, 1, 0, 1], [1, 0, 0, 1], [0, 1, 1, 0]]
# B = [[0, 1], [0, 0], [0, 0], [1, 0]]

prod(A,B)

# [[1, 1], [1, 1], [0, 0]]
```

Example 2.2. Write a function that uses the algorithm of sieve of Eratosthenes to find all the prime numbers up to (and including) the input parameter m.

Recall that this algorithm finds the prime numbers by listing all the integers from 2 to m, and start striking out first any multiples of 2 (e.g., 4, 6, 8,...). After reaching the end of the list, we go back to the beginning, circle 3 (it is prime) and strike out all the multiples of 3, go back to the beginning to the first not removed number (it will be 5), and strike all the multiples of 5, etc. It is important to understand to you need only check the numbers up to $\sqrt{m}$, because of the following lemma.

Lemma 2.3. *Let a be a composite number (i.e., not prime). Then it must have a prime divisor p such that $p \leq \sqrt{a}$.*

I will leave it as an exercise to prove this lemma.

```
def sieve(m):
    '''uses the algorithm sieve of Eratosthenes
    to find all the prime numbers up to m.
    The function returns the list of prime numbers.
    '''

    n = m + 1
    numbers = [True]*n                  #assume all the numbers are prime
    numbers[0] = numbers[1] = False     #0 and 1 are not prime

    for i in range(2, int(n**0.5 + 1)): #we do not need to check all the numbers,
                                        #only up to sqrt(n)
        if numbers[i]:
            for j in range(i*i, n, i):  #a slight improvement, should be checking from i*i,
```

```
                                        #not from 2*i
            numbers[j] = False


    primes = [i for i in range(n) if numbers[i]] #list comprehension!

    return primes

>>>len(sieve(5000000))
348513
```

One new thing here is the line `numbers[0] = numbers[1] = False`, where in one line I assign value `False` to two objects. You can use this syntax to make your code slightly shorter. The output of the function `sieve` gives you the number of prime numbers that do not exceed 5000000.

Example 2.4. Let a be a real number (`int` or `float`) and n be a positive integer. Without using Python's $**$ write a program that computes

$$a^n = a \cdot a \cdot \ldots \cdot a \quad (n \quad times).$$

One obvious solution is to multiply a exactly n times. This solution is given below in function `power_1`. This solution, however, is very slow, especially for large powers. Can we do better? Actually, yes. To understand why start with a special choice for n, namely, powers of 2. How I would compute, e.g., 4^{16}, noticing that $16 = 2^4$? Notice that I can calculate $4^2 = 4 \cdot 4$ first, then $4^4 = 4^2 \cdot 4^2$ (since I already know 4^2), $4^8 = 4^4 \cdot 4^4$ and finally $4^{16} = 4^8 \cdot 4^8$. That is, instead of 15 multiplications, I need only four! How to generalize this to an arbitrary power n? Well, we can represent any integer n as a sum of powers of 2 with coefficients that are either 0 or 1. For instance, if $n = 10$ then

$$10 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

(an experienced reader noticed that I am representing here 10 in base 2).

Now for any a

$$a^{10} = a^{2^3} \cdot a^2,$$

and in general any power can be represented as a product of powers, each of which is a power of two. So the idea is to calculate these powers quickly as described above, and finally multiply all the factors to get the final result. I invite the student carefully look through the function `power_2`, which does exactly what is described in an efficient manner.

To convince you that my second function is way more efficient, I write a test function, which accept as a parameter name of function to be tested, note the simplicity of the syntax for this. I also use package `time` to see how fast my exponentiation works. At the end for comparison I also check how fast built-in operator `**` performs. Please note how close my second function to the built-in in terms of performance.

```
def power_1(a: int, n: int):
    c = a
    for i in range(2,n+1):
        c *=a
    return c
```

```
def power_2(a: int, n: int):
    c = a
    d = 1
    while n != 0:
        if n % 2 == 0:
            n = n // 2
            c = c * c
        else:
            n = n - 1
            d = d * c
    return d

def test(f, a, n, N):
    for i in range(N):
        f(a,n)

import time

t0 = time.time()
test(power_1,23,3147,1500)
t1 = time.time()
print(f"Time required for computation is {t1-t0}")

#Time required for computation is 2.6981542110443115

t0 = time.time()
test(power_2,23,3147,1500)
t1 = time.time()
print(f"Time required for computation is {t1-t0}")

Time required for computation is 0.1760098934173584

t0 = time.time()
for i in range(1500):
       23**3147
t1 = time.time()
print(f"Time required for computation is {t1-t0}")

Time required for computation is 0.10600590705871582
```

2.5 Coding exercises

Your should work through each of the following coding exercises to make sure you are prepared for the next part of the course. Also the coding exercises in the textbook (Section 1.6.1) are highly recommended.

Exercise 1. Let $\boldsymbol{A}$ be an $m \times n$ matrix. Recall that its transpose is $n \times m$ matrix $\boldsymbol{B} = \boldsymbol{A}^\top$ with the elements $b_{ij} = a_{ji}$. Write a function that accepts a matrix $\boldsymbol{A}$ and returns its transpose.

Exercise 2. Recall that Fibonacci numbers are defined as

$$F_n = F_{n-1} + F_{n-2}, \quad n = 2, 3, \ldots, \quad F_0 = 0,\, F_1 = 1.$$

Write a function that returns m-th Fibonacci number.

Exercise 3. Write a function that finds a maximum element of a given list. Do not use any built-in methods. Write another function that returns the index of a maximal element in a given list.

Exercise 4. Using Euclid's algorithm write a function that returns greatest common divisor of two non-negative integers.

Exercise 5. Write a function that checks whether given natural number a is prime.

Exercise 6. Write a program that returns factorization of given natural number n into prime factors.

Exercise 7. For given natural n obtain its decimal representation in the backward order (i.e., if the input 12345 then output should be 54321).

Exercise 8. For a given list of numbers find the number of different elements in this list (without using `set` data structure in Python).

Exercise 9. Write a function that finds the roots of quadratic polynomial

$$ax^2 + bx + c.$$

Consider three possible cases.