DSC 5101 – Computer Programming in DSAI
Part I - Python Programming

HO 02 - Using Function

Opim Salim Sitompul

Department of Data Science and Artificial Intelligence
Universitas Sumatera Utara

Co-funded by the Erasmus+ Programme of the European Union

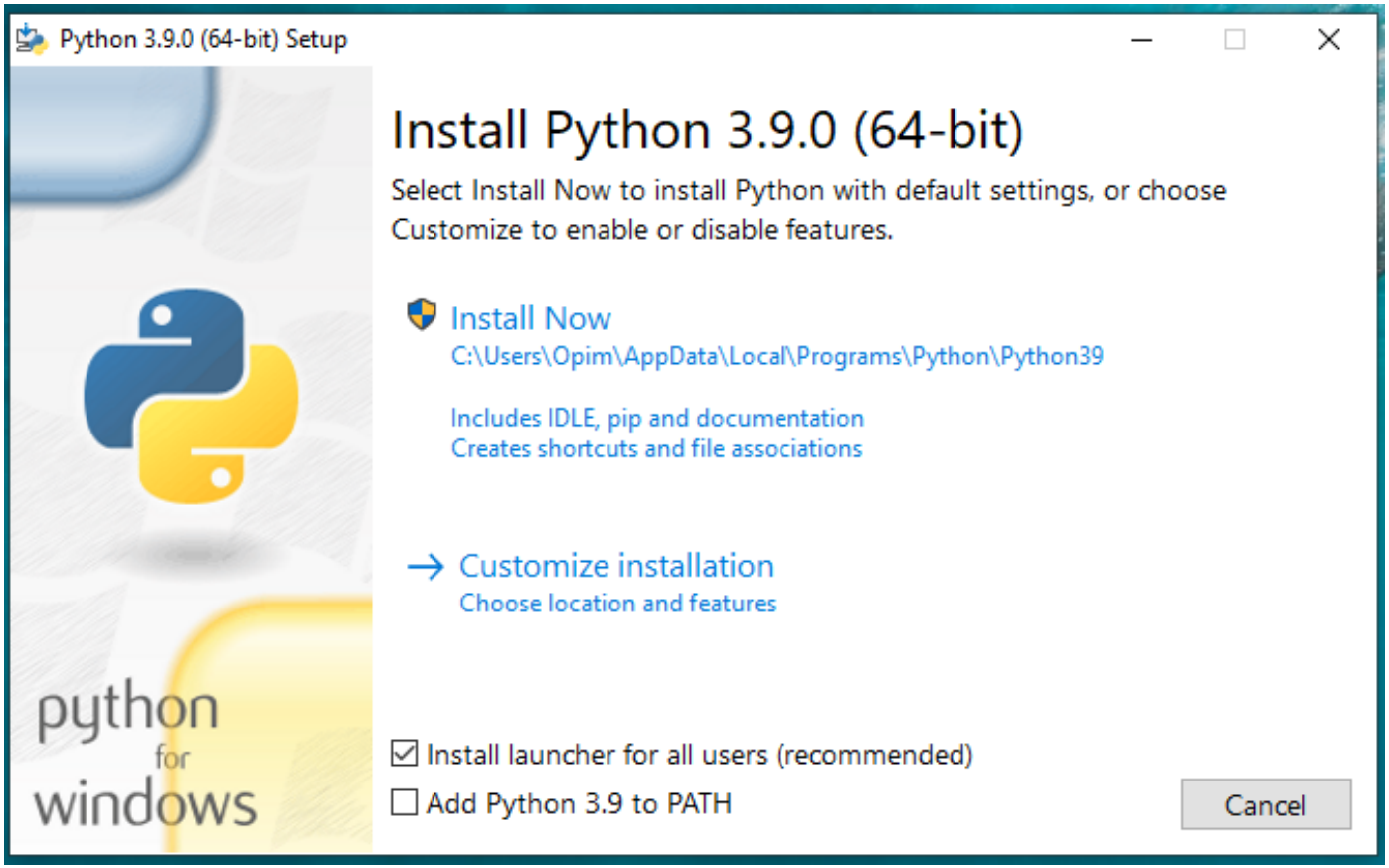
Outline I

Introduction to Python

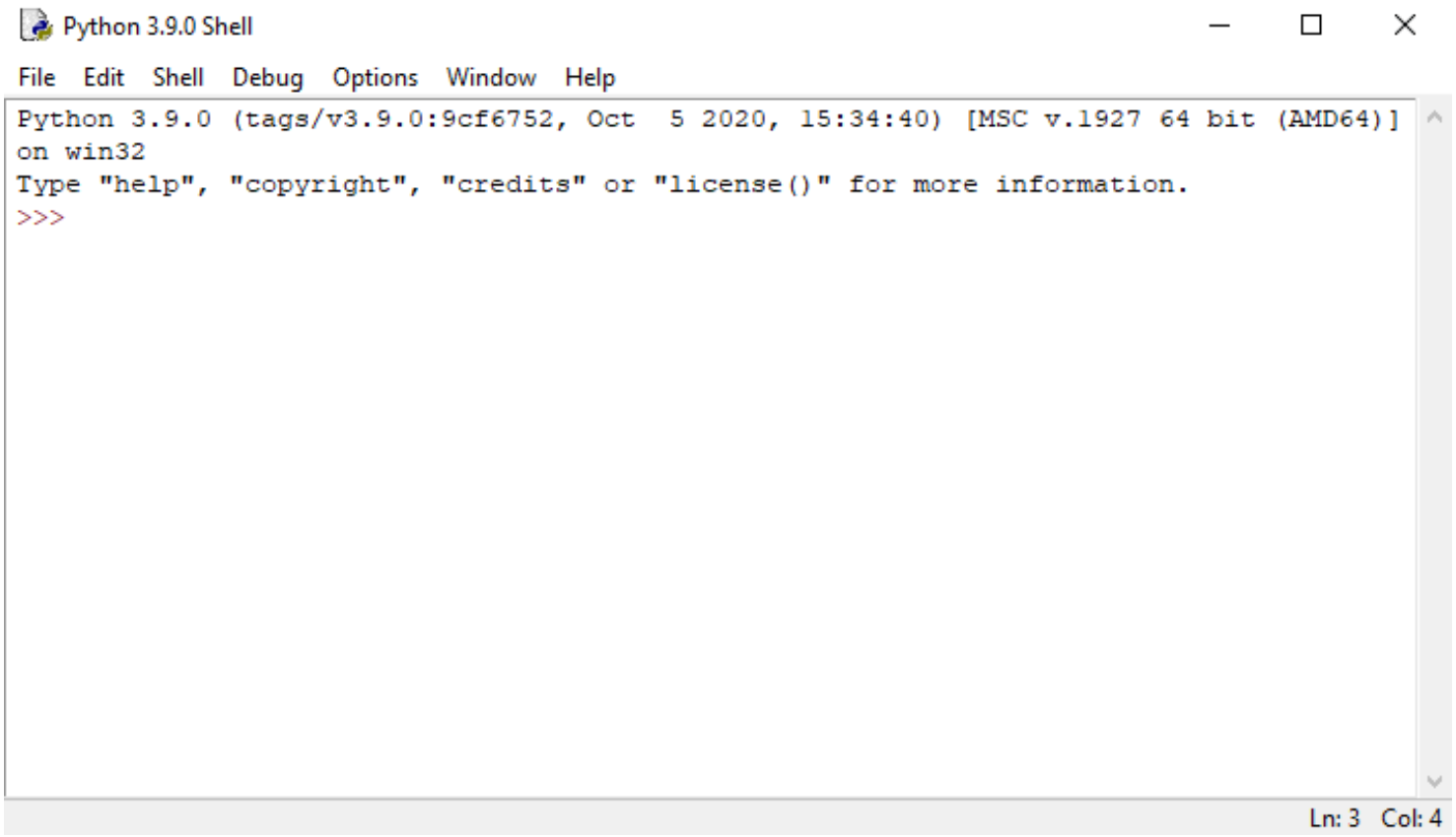
- Python is a first-class tool for scientific computing tasks [1].
- Before we can use Python, we have to consider several options in the installation choices.
  1. Python IDLE
  2. Anaconda
  3. Pycharm
- In this course we will be using Python 3, latest version is Python 3.9.0 (Release Date: Oct. 5, 2020).

Preparing Python using IDLE

- Python IDLE dapat di downloaad pada URL berikut: https://www.python.org/downloads/release/python-390/
- Namafile: python-3.9.0-amd64.exe



Preparing Python using IDLE



Preparing Python Anaconda

- To install Python and the suites of libraries for scientific computing, we can choose one of two Anaconda distributions:
  1. Anaconda (https://www.anaconda.com/products/individual)
  2. Miniconda (https://docs.conda.io/en/latest/miniconda.html)

Preparing Python Anaconda

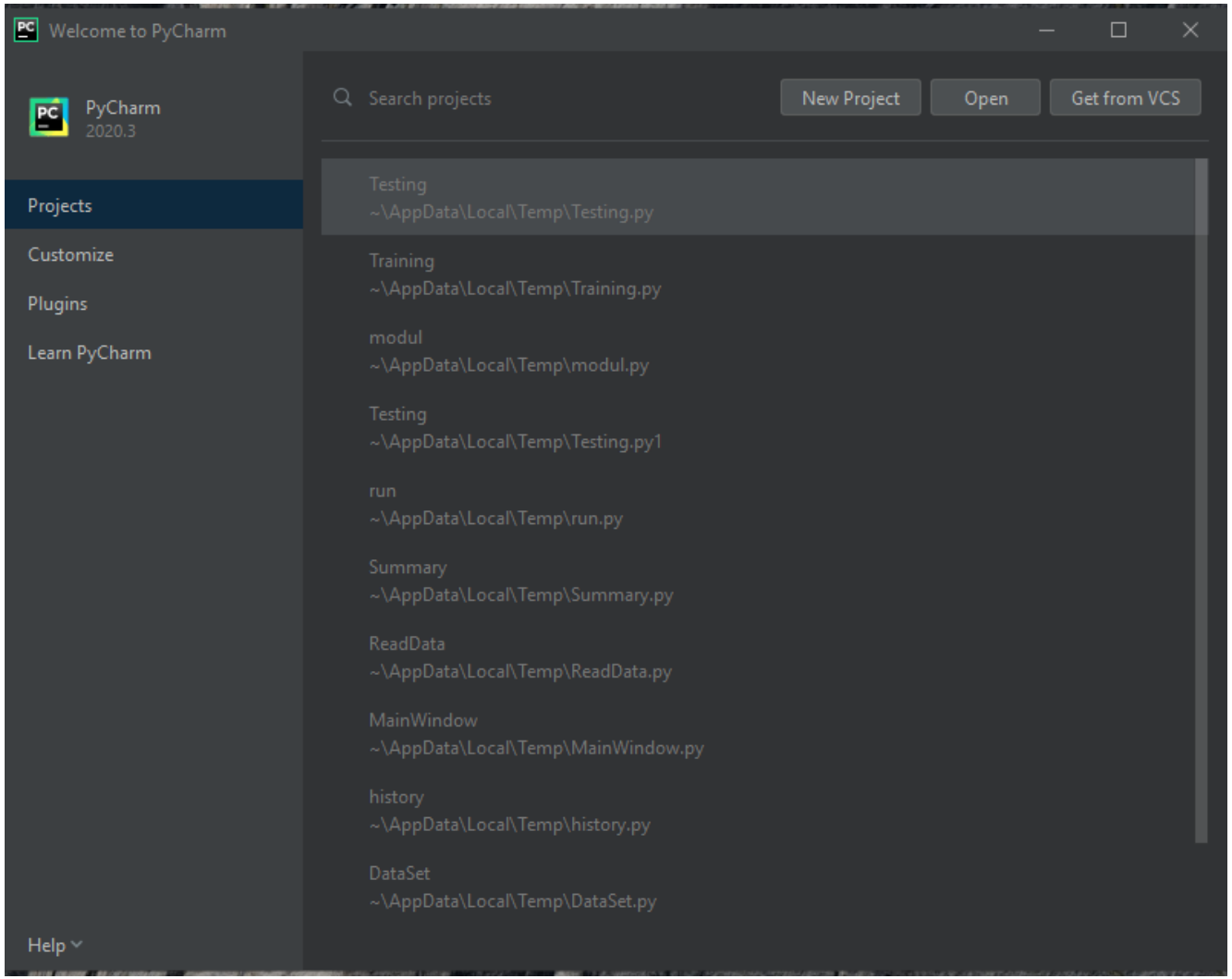
- Choose Anaconda, if you
  - Are new to conda or Python.
  - Like the convenience of having Python and over 1,500 scientific packages automatically installed at once.
  - Have the time and disk space—a few minutes and 3 GB.
  - Do not want to individually install each of the packages you want to use.

Preparing Python Anaconda

- Choose Miniconda, if you
  - Do not mind installing each of the packages you want to use individually.
  - Do not have time or disk space to install over 1,500 packages at once.
  - Want fast access to Python and the conda commands and you wish to sort out the other programs later.

Preparing Python PyCharm

Preparing Python PyCharm



Defining Function

- In this class, we will be using Python functions to organize programs.
- Before that, we will define general form of a Python function.

  def func_name([*params*]):
  |tab| statements

- Keyword **def** will always precede a function definition, followed by the function name, and a list of optional parameters written in parenthesis.
  - The first line of this function definition is called *function header*.
- |tab| is a must indentation to write every statement in a function.

Defining Function

- We will begin with a main function, which will be the first function to be called in a Python program.
- The main function is one special function in Python, which will call other functions written in it such as,

```
def main():
    call_another_function()
```

- When the Python program is executed, the interpreter will check if a function with the name of “main” exist, and call it accordingly, as follows:

```
if __name__ == "__main__":
    main()
```

Defining Function

- Now, let see an example to implement functions.
- What about determining if a number is prime or not?

Defining Function

```
"""
Filename: isprime.py
Determine if a number is prime or not
"""
def checkprime(x):
    rem = 0
    for i in range(2, int(x)):
        if int(x) % i == 0:
            print("Divided by {}".format(i))
            rem = 1
            break
    return rem
```

Defining Function

```
def main():
    x = input("Please give an integer: ")
    if int(x) <= 1:
        print("Smallest prime is 2.")
    else:
        rem = checkprime(x)
        if rem != 0:
            print("{} is not prime".format(x))
        else:
            print("{} is prime.".format(x))

if __name__ == '__main__':
    main()
```

Defining Function

- Some points are worth noted:
  1. Input given to variable *x* is received in a string type, therefore it should be converted into integer, using int(x) (see line 16).
  2. Comparing *x* with a number using $\leq$ could not be done in a string form.
  3. In function *isprime()*, iteration of the **for** loop is terminated as soon as there is a number that divides *x*, using **break** (line 11).
- In order to check what the type of a variable is, we can use *type* command, such as:

  print(type(x))

  Output: <class 'str'>
- We can add one handy function into the *isprime.py* example to handle string input [2], as shown in *isprime2.py*.

Defining Function

```
"""
Filename: isprime2.py
Only allow integer input
"""
def get_integer_input(message):
    """
    If user enters other than numbers, input
        will be rejected
    """
    valstr = input(message)
    while not valstr.isnumeric():
        print("Input must be integer!")
        valstr = input(message)
    return int(valstr)
```

Defining Function

- The complete program will become:

```
def checkprime(x):
    rem = 0
    for i in range(2, x):
        if x % i == 0:
            print("Divided by {}".format(i))
            rem = 1
            break
    return rem
```

Defining Function

```
def main():
    x = get_integer_input("Please give an
        integer: ")
    if x <= 1:
        print("Smallest prime is 2.")
    else:
        rem = checkprime(x)
        if rem != 0:
            print("{} is not prime".format(x))
        else:
            print("{} is prime.".format(x))

if __name__ == '__main__':
    main()
```

Defining Function

1. Function *isnumeric()* called on a string variable *valstr* will check whether the input given is numerical (line 6).
2. The while loop (line 6) will only be entered whenever *valstr* is other than numeric character.
3. Therefore, even though this input is numerical, it still considered as a string and has to be converted from *str* type into integer type (line 9).
4. Note:
   - For example: input of 5 is considered as numeric character and has a value of 54 (in ASCII code), but as an integer 5 has an ordinal value of 5.
5. In the rest of the program lines, the input now is treated as an integer and no more integer conversions are needed (as in lines 23, 16, and 17).

Function Parameters

- There are two kinds of parameters related to function calls and function definitions.
  1. Actual parameter(s) also called argument(s): data passed to a function when a function is called (see line 23 and line 27 of program *isprime2.py*).
  2. Formal parameter(s): parameter(s) written in the function header to receive those data (see line 5 and line 14 of program *isprime2.py*).

Default Parameters

- One function can take zero, one, and more parameters, each of which should be separated by a comma.
- When a program has one or more parameters, it would be possible that some of them might not be used in most cases.
- We can provide default values for some or all of those parameters; particularly for ones which might not be used [2].

Default Parameters

```
"""
Filename: repeatchars.py
Repeatedly displaying characters with default
    parameter
"""

def repeat_chars(ch='*', number=5):
    for i in range(number):
        print(ch, end=' ')
    print()
```

Default Parameters

```
def main():
    repeat_chars()
    repeat_chars('#')
    repeat_chars('%', 15)

if __name__ == '__main__':
    main()
```

Default Parameters

- Function *repeat_chars()* has default values for each of the parameter.
- Function *main()* calls this function three times, giving 0, 1, and 2 parameters, respectively.
- Value given to a parameter will override the default value provided, but if no value is given then the default is used.

Default Parameters

- In terms of default parameters, there are mandatory and optional field, parameter name is mandatory, while the value provided to the parameter is optional.
- Simple rule: if one parameter is given a default value, then all other parameter after this parameter should also have default values.

Named Arguments

- Relying on the position of parameters that have default values are not always satisfying, because it may happens that some of the parameters may need default parameter and some may not.
- In that case we can use named arguments to choose which parameters should be given values.

Named Arguments

```
"""
Filename: display_date.py
Displaying date with named arguments
"""
def mydate(message, d = 1, m=1, y=2000):
    month = get_month_name(int(m))
    date = str(d) + ' ' + month + ' ' + str(y)
    print(message, date)
```

Named Arguments

```
def get_month_name(m):
    if m == 1:
        month = 'January'
    elif m == 2:
        month = 'February'
    elif m == 3:
        month = 'March'
    elif m == 4:
        month = 'April'
    elif m == 5:
        month = 'May'
    elif m == 6:
        month = 'June'
```

Named Arguments

```
    elif m == 7:
        month = 'July'
    elif m == 8:
        month = 'August'
    elif m == 9:
        month = 'September'
    elif m == 10:
        month = 'October'
    elif m == 11:
        month = 'November'
    else:
        month = 'December'
    return month
```

Named Arguments

```

def main():
    mydate("Year of millennium")
    mydate(message="Happy New Year", y=2021)

if __name__ == '__main__':
    main()
```

Named Arguments

- Function *mydate()* has four parameters, three of them are given default values.
- In function *main()* function *mydate()* is called twice, firstly with only one argument, and secondly with named arguments for *message* and *y* parameters.

Arbitrary Arguments

- In the previous definition of a function, the formal arguments list are somewhat fix, even though some of them may have default values.
- However, this is not applicable if we do not know in advance how many arguments will be supplied to the function at the time of defining the function.
- Luckily, we can design a function with arbitrary arguments, using a special argument: **args*.
- Subsequently, we can iterate those arguments in the function body.

Arbitrary Arguments

```
def any_date(*args):
    for date in args:
        one_date = date.split(" ")
        month = get_month_name(int(one_date
            [1]))
        the_date = one_date[0] + ' ' + month +
            ' ' +one_date[2]
        print(the_date, end = ' ')
    print()
```

Arbitrary Arguments

```
def get_month_name(m):
    if m == 1:
        month = 'January'
    elif m == 2:
        month = 'February'
    elif m == 3:
        month = 'March'
    elif m == 4:
        month = 'April'
    elif m == 5:
        month = 'May'
    elif m == 6:
        month = 'June'
```

Arbitrary Arguments

```
    elif m == 7:
        month = 'July'
    elif m == 8:
        month = 'August'
    elif m == 9:
        month = 'September'
    elif m == 10:
        month = 'October'
    elif m == 11:
        month = 'November'
    else:
        month = 'December'
    return month
```

Arbitrary Arguments

```
def main():
    any_date("31 12 1960", "31 12 1970", "31 12 1980", "31 12 1990")
    any_date("1 1 2000", "1 1 2010", "1 1 2020")

if __name__ == '__main__':
    main()
```

Arbitrary Arguments

- In *any_date()*, arbitrary number of arguments are received by **args*.
- These arguments are iteratively separated using a **for** loop where each argument is saved in *date*.
- Using *split()* on string variable *date*, the string are split into three slices based on character " ", contained in this string, resulted in a list with three members: date[0], date[1], and date[2].
- Function *main* calls *any_date()* twice with four and three arguments on each call.

Positional and Keyword Arguments

- When a function is called with some values, these values get assigned to the arguments exactly according to their position, no more no less.
- To be free from these positional arguments, formal arguments of a function can also defined further using positional (*args*) and keyword (*kwargs*) arguments.
- Using keyword argument, the order of the arguments can be changed, the same principle used as in named arguments, but in this case with arbitrary number of arguments.

Anonymous Function

- For a simple function definition containing one line of instruction, we can define an anonymous function.
- The function is defined using **lambda** keyword.

  lambda arguments: expression

Anonymous Function

- When an anonymous (lambda) function is called within some special function such as *filter()* and *map()*, the lambda function is very useful.
- Function *map()* takes each of list element and passes it to **lambda** to be further processed.
- Function *filter()* will filter the list elements resulted from a **lambda** function.

Anonymous Function

- For example:

  some_list = list(map(lambda x: checkprime(x), a_list))

  zero_list = list(filter(lambda x: x == 0, some_list))
- Function *map()* will take each element of a_list to be check whether it is a prime number using **lambda**.
- Function *filter()* will select only element of zero from *some_list* processed by **lambda** and put them into *zero_list*.

Anonymous Function

```
"""
Filename: lambda_fun.py
Demonstrate the use of lambda function
"""
import random

def create_random(count):
    rand_list = []
    for i in range(count):
        r = random.randint(0, 100)
        rand_list.append(r)

    return(rand_list)
```

Anonymous Function

```
def checkprime(x):
    rem = 0
    if int(x) > 2:
        for i in range(2, int(x)):
            if int(x) % i == 0:
                rem = 1
                break
    elif int(x) == 2:
        rem = 0
    else:
        rem = 1
    return rem
```

Anonymous Function

```
def main():
    one_list = create_random(10)
    prim_list = list(map(lambda x: checkprime(x), one_list))
    ct = 0
    for prim in prim_list:
        if prim == 0:
            print("{} is prime".format(one_list[ct]))
        else:
            print("{} is not prime".format(one_list[ct]))
        ct += 1

if __name__ == '__main__':
    main()
```

Anonymous Function

- The program is begun with a statement to import *random* library class.
- Using *randint* the *create_random()* function iteratively create random integer number in range of 0 to 100 and put each of them into a *rand_list*.
- In function *main()* the *create_random* function is called by submitting a number 10 and save the result into *one_list*.
- The *lambda* function uses *map* function to check each element of *one_list* whether it is a prime number, by calling the *checkprime()* function.

Importing External Function

- External functions are often provided by third parties, especially to be used in special cases.
- External functions are usually packaged into a class.
- In order to use those functions in a program, we have to **import** them into our code program.
- The following example illustrate the use of function *random()* in order to generate pseudo random numbers.
- In our program, we will include those function using **import** random.

Importing External Function

```
import random

def create_random_array(num_arr):
    myArr = []
    for i in range(num_arr):
        myArr.append(random.randint(1,num_arr)
            )
    return myArr

def print_arr(num_arr):
    for num in num_arr:
        print("{} ".format(num), end=' ')

def main():
    x = input("Please give an integer: ")
    my_arr = create_random_array(int(x))
```

References I

[1] VanderPlas J., Python Data Science Handbook: Essential Tools for Working with Data, O'Reilly, USA, 2016.

[2] Hunt, J., A Beginners Guide to Python 3 Programming, Springer Nature Switzerland AG, Switzerland, 2019.

[3] Hunt, J., Advance Guide to Python 3 Programming, Springer Nature Switzerland AG, Switzerland, 2019.