# Python Programming and Numerical Methods

Ismail JAMIAI (email: ijamiai@uae.ac.ma)

\today

## 1 Chapter 1: Getting Started with Python

### 1.1 1.0. Introduction

Simplicity and readability along with a large number of third-party packages have made Python the "go-to" programming language for numerical methods. Given such an important role of Python for numerical methods, this chapter provides a quick introduction to the language. The introduction is mainly geared toward those who have no knowledge of Python programming but are familiar with programming principles in another (object oriented) language; for example, we assume readers are familiar with the purpose of binary and unary operators, repetitive and conditional statements, functions, a class (a recipe for creating an object), object (instance of a class), method (the action that the object can perform), and attributes (properties of class or objects).

### 1.2 1.1. First Things First: Installing What Is Needed

Install Anaconda! Anaconda is a data science platform that comes with many things that are required throughout the book. It comes with a Python distribution, a package manager known as conda, and many popular data science packages and libraries such as NumPy, pandas, SciPy, scikit-learn, and Jupyter Notebook. This way we don't need to install them separately and manually (for example, using "conda" or "pip" commands). To install Anaconda, refer to (Anaconda, 2023). There are also many tutorials and videos online that can help installing that

### 1.3 1.2. Jupyter Notebook

Python is a programming language that is interpreted rather than compiled; that is, we can run codes line-by-line. Although there are several IDEs (short for, integrated development environment) that can be used to write and run Python codes, the IDE of choice in this book is Jupter notebook. Using Jupyter notebook allows us to write and run codes and, at the same time, combine them with text and graphics. In fact all materials for this book are developed in Jupyter notebooks. Once Anaconda is installed, we can launch Jupyter notebook either from Anaconda Navigator panel or from terminal.

In a Jupyter notebook, everything is part of cells. Code and markdown (text) cells are the most common cells. For example, the text we are now reading is written in a text cell in a Jupyter notebook. The following cell is a code cell and will be presented in gray boxes throughout the book. To run the contents of a code cell, we can click on it and press "shift + enter".

Try it on the following cell and observe a 7 as the output:

```
2 + 5
```

• First we notice that when the above cell is selected in a Jupyter notebook, a rectangle with a green bar on the left appears. This means that the cell is in "edit" mode so we can write codes in that cell (or simply edit if it is a Markdown cell). • Now if "esc" key (or "control + M") is pressed, this bar turns blue. This means that the cell is now in "command" mode so we can edit the notebook as a whole without typing in any individual cell. • Also depending on the mode, different shortcuts are available. To see the list of shortcuts for different modes, we can press "H" key when the cell is in command mode; for example, some useful shortcuts in this list for command mode are: "A" key for adding a cell above the current cell; "B" key for adding a cell below the current cell; and pressing "D" two times to delete the current cell.

## 1.4 1.3. Variables

Consider the following code snippet:

```
x=2.2       # this is a comment ( use "#" for comments )
y=2
x*y
```

In the above example, we created two scalar variables \(x \) and \(y \), assigned them to some values, and multiplied. There are different types of scalar variables in Python that we can use:

```
x = 1                       # an integer
x = 0.3                     # a floating-point
x = "what a nice day"       # a string
x = True                    # a boolean variable (True or False)
x = None                    # None type (the abscence of any value)
```

In the above cell, a single variable \(x \) refers to an integer, a floating-point, string, etc. This is why Python is known as being dynamically-typed; that is, we do not need to declare variables like what is done in C (no need to specify their types) or even they do not need to always refer to the objects of the same type—and in Python everything is an object! This is itself a consequence of the fact that Python variables are references! In Python, an assignment statement such as \( x = 1 \), creates a reference \( x \) to a memory location storing object \( 1 \) (yes, even numbers are objects!). At the same, types are attached to the objects on the right, not to the variable name on the left. This is the reason that \( x \) could refer to all those values in the above code snippet. We can see the type of a variable (the type of what is referring to) by type() method.

For example,

```
x = 3.3
display(type(x))

x = True
display(type(x))

x = None
display(type(x))
```

```
bool
```

```
NoneType
```

Here we look at an attribute of an integer to show that in Python even numbers are objects:

```
(4).imag
```

The imag attribute extracts the imaginary part of a number in the complex domain representation. The use of parentheses though is needed here because otherwise, there will be a confusion with floating points.

**The multiple use of display() in the above code snippet is to display multiple desired outputs in the cell. Remove the display() methods (just keep all type(x)) and observe that only the last output is seen. Rather than using display() method, we can use print() function. Replace display() with print() and observe the difference.**

## 1.5 1.4. Strings

A string is a sequence of characters. We can use quotes (‘ ’) or double quotes (“ ”) to create strings. This allows using apostrophes or quotes within a string. Here are some examples:

```
string1 = "This is a string"
string1
```

```
string2 = "Well, this is 'string' too!"
string2
```

```
string3 = "Ahmed said: 'How are you?'"
string3
```

Concatenating strings can be done using (+) operator:

```
string4 = string1 + ". " + string2
string4
```

We can use \t and \n to add tab and newline characters to a string, respectively:

```
string = "Here we use a newline\n to go to the next\t line"
string
```

## 1.6 1.5. Some Important Operators

### 1.6.1 1.5.1 Arithmetic Operators

The following expressions include arithmetic operators in Python:

```
x = 5
y = 2
 x + y       # addition
# x - y       # substraction
# x * y       # multiplication
# x / y       # division
# x // y      # floor division (removing fractional parts)
x % y       # modulus (integer remanider of division)
# x ** y      # x to the power of y
```

If in the example above, we change \( x = 5 \) to \( x = 5.1 \), then we observe some small rounding errors. Such unavoidable errors are due to internal representation of floating-point numbers.

### 1.6.2 1.5.2 Relational and Logical Operators

The following expressions include relational and logical operators in Python:

```
# x < 5               # less than (> is greater than)
# x <= 5       # less than or equal to (>= is greater than or equal to)
# x == 5              # equal to
# x != 5              # not equal to
# (x > 4 and y < 3)   # "and" keyword is used for logical and
# (x < 4 or y > 3)    # "or" keyword is used for logical or
# (not x > 4)         # "not" keyword is used for logical not
```

### 1.6.3 1.5.3 Membership Operators

Membership operator is used to check whether an element is present within a collection of data items. By collection, we refer to various (ordered or unordered) data structures such as string, lists, sets, tuples, and dictionaries, which are discussed in Section 2.6. Below are some examples:

```
"Hello" in "HelloWorlds!"
```

```
"Hello" in "HellOWorlds!"
```

```
320 in ["Hi", 320, False, "Hello"]
```

## 1.7 1.6. Built-in Data Structures

Python has a number of built-in data structures that are used to store multiple data items as separate entries.

### 1.7.1 1.6.1 Lists

Perhaps the most basic collection is a list. A list is used to store a sequence of objects (so it is ordered). It is created by a sequence of comma-separated objects within [ ]:

```
x = [5, 3.20, 10, 200.2]
x[0]     # the index starts from 0
```

Lists can contain objects of any data types:

```
x = ["Jupyther", 75, None, True, [34, False], 2, 75]  # observe that this list
↪ contains another list
x[4]
```

In addition, lists are mutable, which means they can be modified after they are created. For example,

```
x[4] = 52   # here we change one element of list x
x
```

We can use a number of functions and methods with a list:

```
len(x) # here we use a built-in function to return the length of a list
```

```
y = [9, 0, 4, 2]
x + y
# y * 3
```

```
z = [y, x]
z
```

**Accessing the elements of a list by indexing and slicing:** We can use indexing to access an element within a list (we already used it before):

```
x[3]
```

To access the elements of nested lists (list of lists), we need to separate indices with square brackets:

```
z[1][0]  # this way we access the second element within z and within that we
↪ access the first element
```

A negative index has a meaning:

```
x[-1]  # index -1 returns the last item in the list; -2 returns the second item
↪ from the end, and so forth
```

*Slicing* is used to access multiple elements in the form of a sub-list. For this purpose, we use a colon to specify the start point (**inclusive**) and end point (**non-inclusive**) of the sub-list. For example:

```
x[0:4] # the last element seen in the output is at index 3
```

In this format, if we don't specify a starting index, Python starts from the beginning of the list:

```
x[:4] # equivalent to x[0:4]
```

Similarly, if we don't specify an ending index, the slicing includes the end of the list:

```
x[4:]
```

Negative indices can also be used in slicing:

```
x
```

```
x[-2:]
```

```
x[4::-2]  # a negative step returns items in reverse (it works backward so here we start from the element at index 4 and go backward to the beginning with steps of 2)
```

In the above example, we start from 4 but because the stride is negative, we go backward to the beginning of the list with steps of 2. In this format, when the stride is negative and the "stop" is not specified, the "stop" becomes the beginning of the list. This behavior would make sense if we observe that to return elements of a list backward, the stopping point should always be less than the starting point; otherwise, an empty list would have been returned.

**Modifying elements in a list:** As we saw earlier, one way to modify existing elements of a list is to use indexing and assign that particular element to a new value. However, there are other ways we may want to use to modify a list; for example, to append a value to a list or to insert an element at a specific index. Python has some methods for these purposes. Here we present some examples to show these methods:

```
x.append(-33)  # to append a value to the end of the list
x
```

```
y.sort()  # to sort the element of y
y
```

```
x.insert(2, 10)  # insert(pos, elmnt) method inserts the specified
x
```

```
x.pop(3) # pop(pos) method removes (and returns) the element at the specified position
x
```

```
del x[1] # del statement can also be used to delete an element from a list by its index
x
```

```
x.pop() # by default the position is -1, which means that it removes the last element
x
```

**Copying a List:** It is often desired to make a copy of a list and work with it without affecting the original list. In these cases if we simply use the assignment operator, we end up changing the original list! Suppose we have the following list:

```
list1 = ['A+', 'A', 'B', 'C+']
```

```
list2 = list1
list2
```

```
list2.append('D')
list1
```

As seen in the above example, when 'D' is appended to **list2**, it is also appended at the end of list1. One way to understand this behavior is that when we write **list2 = list1**, in fact what happens internally is that variable **list2** will point to the same container as list1. So if we modify the container using **list2**, that change will appear if we access the elements of the container using list1.
There are three simple ways to properly copy the elements of a list:

1. slicing;
2. copy() method;
3. the list() constructor.

They all create shallow copies of a list (in contrast with deep copies). Based on Python documentation Python copy (2024).

Further implications of these statements are beyond our scope and readers are encouraged to see other references. Here we examine these approaches to create list copies.

```
list3 = list1[:] # the use of slicing; that is, using [:] we make a shallow copy of the entire list1
list3.append("E")
list3
```

```
list1
```

As we observe in the above example, list1 and list3 are different—changing some elements in the copied list did not change the original list (this is in contrast with slicing NumPy arrays that we will see in Chapter 2). Next, we examine the copy method:

```
list4 = list1.copy() # the use of copy() method
list4.append("E")
list4
```

```
list1
```

And finally, we use the list() constructor:

```
list5 = list(list1) # the use of list() constructor
list5.append("E")
list5
```

```
list1
```

help() is a useful function in Python that shows the list of attributes/methods defined for an object. For example, we can see all methods applicable to list1 (for brevity part of the output is omitted):

```
help(list1)
```

### 1.7.2 1.6.2 Tuples 13

Tuple is another data-structure in Python that similar to list can hold other arbitrary data types. However, the main difference between tuples and lists is that a tuple is immutable; that is, once it is created, its size and contents can not be changed.
A tuple looks like a list except that to create them, we use parentheses ( ) instead of square brackets [ ]:

```
tuple1 = ('Machine', 'Learning', 'with', 'Python', '1.0.0')
tuple1
```

Once a tuple is created, we can use indexing and slicing just as we did for a list (using square brackets):

```
tuple1[0]
```

```
tuple1[::2]
```

```
len(tuple1) # the use of len() to return the length of tuple
```

An error is raised if we try to change the content of a tuple:

```
tuple1[0] = "Jupyter" # Python does not permit changing the value
```

Because we can not change the contents of tuples, there is no append or remove method for tuples. Although we can not change the contents of a tuple, we could redefine our entire tuple (assign a new value to the variable that holds the tuple):

```
tuple1 = ("Jupyter", "NoteBook") # redefine tuple1
tuple1
```

Or if desired, we can concatenate them to create new tuples:

```
tuple2 = tuple1 + ("Good", "Morning")
tuple2
```

A common use of tuples is in functions that return multiple values. For example, modf() function from math module (more on functions and modules later), returns a two-item tuple including the fractional part and the integer part of its input:

```
from math import modf # more on "import" later. For now just read this as "from
↪  math module, import modf function" so that modf function is available in our
↪  program
a = 56.5
modf(a) # the function is returning a two-element tuple
```

We can assign these two return values to two variables as follows (this is called sequence unpacking and is not limited to tuples):

```
x, y = modf(a)
"x = " + str(x) + "\n" + "y = " + str(y)
```

Now that we discussed sequence unpacking, let us examine what sequence packing is. In Python, a sequence of comma separated objects without parentheses is packed into a tuple. This means that another way to create the above tuple is:

```
tuple1= 'Machine', 'Learning', 'with', 'Python', '1.0.0' # sequence packing
tuple1
```

```
x, y, z, v, w = tuple1 # the use of sequence unpacking
(x, y, z, v, w)
```

Note that in the above example, we first packed 'Machine', 'Learning', 'with', 'Python', '1.0.0' into tuple1 and then unpacked into x, y, z, v, w. Python allows to do this in one step as follows (also known as multiple assignment, which is really a combination of sequence packing and unpacking):

```
x, y, z, v, w = 'Machine', 'Learning', 'with', 'Python', '1.0.0'
(x, y, z, v, w)
```

We can do unpacking with lists if desired:

```
list6 = ['Machine', 'Learning', 'with', 'Python', '1.0.0']
x, y, z, v, w = list6
(x, y, z, v, w)
```

One last note about sequence packing for tuples: if we want to create a one-element tuple, the comma is required (why?):

```
tuple3 = 'Machine', # remove the comma and see what would be the type here
type(tuple3)
```

### 1.7.3 1.6.3 Dictionaries

A dictionary is a useful data structure that contains a set of values where each value is labeled by a unique key (if we duplicate keys, the second value wins). We can think of a dictionary data type as a real dictionary where the words are keys and the definition of words are values but there is no order among keys or values. As for the keys, we can use any immutable Python built-in type such as string, integer, float, boolean or even tuple as long the tuple does not include a mutable object. In technical terms, the keys should be hashable but the details of how a dictionary is implemented under the hood is out of the scope here.
Dictionaries are created using a collection of key:value pairs wrapped within curly braces { } and are non-ordered:

```
dict1 = {
    1:'value for key 1',
    'key for value 2':2,
    (1,0):True,
    False:[100,50],
    2.5:'Hello'
}
dict1
```

The above example is only for demonstration to show the possibility of using immutable data types for keys; however, keys in dictionary are generally short and more uniform. The items in a dictionary are accessed using the keys:

```
dict1['key for value 2']
```

Here we change an element:

```
dict1['key for value 2'] = 30 # change an element
dict1
```

We can add new items to the dictionary using new keys:

```
dict1[10] = 'Bye'
dict1
```

del statement can be used to remove a key:pair from a dictionary

```
del dict1['key for value 2']
dict1
```

As we said, a key can not be a mutable object such as list:

```
dict1[["1","(1,0)"]] = 100 # list is not allowed as the key
```

The non-ordered nature of dictionaries allows fast access to its elements regardless of its size. However, this comes at the expense of significant memory overhead (because internally uses an additional sprase hash tables). Therefore, we should think of dictionaries as a trade off between memory and time: as long as they fit in the memory, they provide fast access to their elements.

In order to check the membership among keys, we can use the keys() method to return a dict_keys object (it provides a view of all keys) and check the membership:

```
(1,0) in dict1.keys()
```

This could also be done with the name of the dictionary (the default behaviour is to check the keys, not the values):

```
(1,0) in dict1 # equivalent to: in dict1.keys()
```

In order to check the membership among values, we can use the values() method to return a dict_values object (it provides a view of all values) and check the membership:

```
"Hello" in dict1.values()
```

Another common way to create a dictionary is to use the dict() constructor. It works with any iterable object (as long each element is iterable itself with two objects), or even with comma separated key=object pairs. Here is an example in which we have a list (an iterable) where each element is a tuple with two objects:

```
dict2 = dict([("Police", 102), ("Fire", 101), ("Gas", 104)])
dict2
```

Here is another example in which we have pairs of comma-separated key=object:

```
dict3 = dict(Country='USA', phone_numbers=dict2, population_million=18.7) # the
↪  use of keywords arguments = object
dict3
```

### 1.7.4 1.6.4 Sets

Sets are collection of non-ordered unique and immutable objects. They can be defined similarly to lists and tuples but using curly braces. Similar to mathematical set operations, they support union, intersection, difference, and symmetric difference. Here we define two sets, namely, set1 and set2 using which we examine set operations:

```
set1 = {"a", "b", "c", "d", "e"}
set1
```

```
set2 = {"b", "b", "c", "f", "g"}
set2 # observe that the duplicate entery is removed
```

```
set1 | set2 # union using an operator. Equivalently, this could be done by
↪  set1.union(set2)
```

```
set1 & set2 # intersection using an operator. Equivalently, this could be done by
↪  set1.intersection(set2)
```

```
set1 - set2 # difference: elements of set1 not in set2. Equivalently, this could
↪  be done by set1.difference(set2)
```

```
set1 ^ set2 # symmetric difference: elements only in one set, not in both.
↪  Equivalently, this could be done by set1.symmetric_difference(set2)
```

```
"b" in set1 # check membership
```

We can use help() to see a list of all available set operations:

```
help(set1) # output not shown
```

### 1.7.5 1.6.5 Some Remarks on Sequence Unpacking

Consider the following example

```
x, *y, v, w = ['Machine', 'Learning', 'with', 'Python', '1.0.0']
(x, y, v, w)
```

As seen in this example, variable y becomes a list of 'Learning' and 'with'. Note that the value of other variables are set by their positions. Here * is working as an operator to implement extended iterable unpacking. We will discuss what an iterable or iterator is more formally in Section 1.7.1 and Section 1.9.1 but, for the time being, it suffices to know that any list or tuple is an iterable object. To better understand the extended iterable unpacking, we first need to know how * works as the iterable unpacking.
We may use * right before an iterable in which case the iterable is expanded into a sequence of items, which are then included in a second iterable (for example, list or tuple) when unpacking is performed. Here is an example in which * operates on an iterable, which is a list, but at the site of unpacking, we create a tuple.

```
*[1, 2, 3], 5
```

Here is a similar example but this time the "second" iterable is a list:

```
[*[1, 2, 3], 5]
```

We can also create a set:

```
{*[1, 2, 3], 5}
```

Now consider the following example:

```
*[1, 2, 3]
```

SyntaxError: can't use starred expression here

The above example raises an error. This is because iterable unpacking can be only used in certain places. For example, it can be used inside a list, tuple, or set. It can be also used in list comprehension (discussed in Section 1.7.2) and inside function definitions and calls (more on functions in Section 1.8.1). A mechanism known as extended iterable unpacking that was added in Python 3 allows using * operator in an assignment expression; for example, we can have statements such as a, b, *c = some-sequence or *a, b, c = some_sequence. This mechanism makes * as "catch-all"

operator; that is to say, any sequence item that is not assigned to a variable is assigned to the variable following * . That is the reason that in the example presented earlier, y becomes a list of 'Learning' and 'with' and the value of other variables are assigned by their position. Furthermore, even if we change the list on the right to a tuple, y is still a list:

```
x, *y, v, w = ('Machine', 'Learning', 'with', 'Python', '1.0.0')
(x, y, v, w)
```

To create an output as before (i.e., Machine Learning with Python 1.0.0), it suffices to use again * before y in the print function; that is,

```
(x, *y, v, w)
```

## 1.8 1.7. Flow of Control and Some Python Idioms

### 1.8.1 1.7.1 for Loops

for loop statement in Python allows us to loop over any iterable object. What is a Python iterable object? An iterable is any object capable of returning its members one at a time, permitting it to be iterated over in a for loop. For example, any sequence such as list, string, and tuple, or even non-sequential collections such as sets and dictionaries are iterable objects. To be precise, to loop over some iterable objects, Python creates a special object known as iterator (more on this in Section 1.9.1); that is to say, when an object is iterable, under the hood Python creates the iterator object and traverses through the elements. The structure of a for loop is quite straightforward in Python:

```
for variable in X:
    the body of the loop
```

In the above representation of the for loop sturcture, \(X\) should be either an iterator or an iterable (because it can be converted to an iterator object). It is important to notice the indentation. Yes, in Python indentation is meaningful! Basically, code blocks in Python are identified by indentation. At the same time, any statement that should be followed by an indented code block is followed by a colon : (notice the : before the body of the loop). For example, to iterate over a list:

```
  for x in list1:
      print(x)
```

Iterate over a string:

```
  string = "Hi There"
  for x in string:
      print(x, end = "")
```

Iterate over a dictionary:

```
dict2 = {1:"machine", 2:"learning", 3:"with python"}
for key in dict2: # looping through keys in a dictionary
    val = dict2[key]
    print('key =', key)
    print('value =', val)
    print()
```

As we mentioned in Section 1.6.3, in the above example, we can replace dict2 with dict2.keys() and still achieve the same result:

```
for key in dict2.keys(): # looping through keys in a dictionary
    val = dict2[key]
    print('key =', key)
    print('value =', val)
    print()
```

When looping through a dictionary, it is also possible to fetch the keys and values at the same time. For this purpose, we can use the items() method. This method returns a dict_items object, which provides a view of all (key, value) tuples in a dictionary. Next we use this method along with sequence unpacking for a more Pythonic implementation of the above example. A code pattern is generally referred to as Pythonic if it uses patterns known as idioms, which are in fact some code conventions acceptable by the Python community (e.g., the sequence packing and unpacking we saw earlier were some idioms):

```
for key, val in dict2.items():
    print("key=", key)
    print("value=", val)
    print()
```

Often we need to iterate over a sequence of numbers. In these cases, it is handy to use range() constructor that returns a range object, which is an iterable and, therefore, can be used to create an iterator needed in a loop. These are some common ways to use range():

```
for i in range(5): # the sequence from 0 to 4
    print("i =", i)
```

```
for i in range(3, 8): # the sequence from 3 to 7
    print("i =", i)
```

```
for i in range(3, 8, 2): # the sequence from 3 to 7 with steps 2
    print("i =", i)
```

When looping through a sequence, it is also possible to fetch the indices and their corresponding values at the same. The Pythonic way to do this is to use enumerate(iterable, start=0), which returns an iterator object that provides the access to indices and their corresponding values in the form of a two-element tuple of index-value pair:

```
for i, v in enumerate(tuple1):
    print(i, v)
```

Compare this simple implementation with the following (non-Pythonic) way to do the same task:

```
i = 0
for v in tuple1:
    print(i, v)
    i += 1
```

Here we start the count from 1:

```
for i, v in enumerate(tuple1, start=1):
    print(i, v)
```

Here we use enumerate() on a tuple:

```
for i, v in enumerate(tuple1):
    print(i, v)
```

**enumerate()** can also be used with sets and dictionaries but we have to remember that these are non-ordered collections so in general it does not make much sense to fetch an index unless we have a very specific application in mind (e.g., sort some elements first and then fetch the index). Another example of a Python idiom is the use of **zip()** function, which creates an iterator that aggregates two or more iterables, and then loops over this iterator.

```
list_a = [1, 2, 3, 4]
list_b = ["a", "b", "c", "d"]
for item in zip(list_a, list_b):
    print(item)
```

We mentioned previously that one way to create dictionaries is to use the dict() constructor, which works with any iterable object as long as each element is iterable itself with two objects. Assume we have a name_list of three persons, John, James, Jane. Another list called phone_list contains their numbers that are 979, 797, 897 for John, James, and Jane, respectively. We can now use dict() and zip to create a dictionary where keys are names and values are numbers:

```
name_list = ["Ahmed", "Samir", "Nouh"]
phone_list = [611, 790, 638]
dict3 = dict(zip(name_list, phone_list))
dict3
```

### 1.8.2 1.7.2 List Comprehension

Once we have an iterable, it is often required to perform three operations: • select some elements that meet some conditions; • perform some operations on every element; and • perform some operations on some elements that meet some conditions.

Python has an idiomatic way of doing these, which is known as list comprehension (short form listcomps). The name list comprehension comes from mathematical set comprehension or abstraction in which a set is defined based on the properties of its members. Suppose we would like to create a list containing square of odd numbers between 1 to 20. A non-Pythonic way to do this is:

```
list_odd = []
for i in range(1, 21):
    if i%2 != 0:
        list_odd.append(i**2)

list_odd
```

List comprehension allows us to combine all this code in one line by combining the list creation, appending, the for loop, and the condition:

```
list_odd_lc = [i**2 for i in range(1, 21) if i%2 != 0]
list_odd_lc
```

list_odd_lc is created from an expression (i**2) within the square brackets. The numbers fed into this expression comes from the for and if clause following the expression (note that there is no ":" (colon) after the for or if statements). Observe the equivalence of this list comprehension with the above "non-Pythonic" way to properly interpret the list comprehension. The general syntax of applying listcomps is the following:

[expression for exp_1 in seq_1 if condition_1 for exp_2 in seq_2 if condition_2 — for exp_n in seq_n if condition_n

]

Here is another example in which we use the list comprehension to generate a list of two-element tuples of non-equal integers between 0 and 3:

```
list_non_equal_tuples = [(x, y) for x in range(3) for y in range(3) if x != y]
list_non_equal_tuples
```

### 1.8.3 1.7.3 if-elif-else

Conditional statements can be implemented by if-elif-else statement:

```
    list4 = ["Machine", "Learning", "with", "Python"]
    if "java" in list4:
        print("There is java too!")
    elif "C++" in list4:
        print("There is C++ too!")
    else:
        print("Well, just Python there.")
```

In these statements, the use of **else** or **elif** is optional.

## 1.9 1.8. Function, Module, Package, and Alias

### 1.9.1 1.8.1 Functions

Functions are simply blocks of code that are named and do a specific job. We can define a function in Python using def keyword as follows:

```
def substract_three_numbers(num1, num2, num3):
    result = num1 - num2 - num3
    return result
```

In the above code snippet, we define a function named subtract_three_numbers with three inputs num1, num2, and num3, and then we return the result. We can use it in our program, for example, as follows:

```
x = substract_three_numbers(10, 3.0, 2)
print(x)
```

In the above example, we called the function using positional arguments (also sometimes simply referred to arguments); that is to say, Python matches the arguments in the function call with the parameters in the function definition by the order of arguments provided (the first argument with the first parameter, second with second, . . . ). However, Python also supports keyword arguments in which the arguments are passed by the parameter names. In this case, the order of keyword arguments does not matter as long as they come after any positional arguments (and note that the definition of function remains the same). For example, this is a valid code:

```
x = substract_three_numbers(num3 = 2, num1 = 10, num2 = 3.0)
print(x)
```

but substract_three_numbers(num3 = 1, num2 = 3.0, 10) is not legitimate because 10 (positional argument) follows keyword arguments.

In Python functions, we can return any data type such as lists, tuples, dictionaries, etc. We can also return multiple values. They are packed into one tuple and upon returning to the calling environment, we can unpack them if needed:

```
def string_func(string):
    return len(string), string.upper(), string.title()

string_func("coolFunctions") # observe the tuple
```

```
x, y, z = string_func("coolFunctions")
print(x, y, z)
```

If we pass an object to a function and within the function the object is modified (e.g., by calling a method for that object and somehow change the object), the changes will be permanent (and nothing need to be returned):

```
def list_mod(inp):
    inp.insert(1, "AB")
```

```
3
4   list7 = [100, "ML", 200]
5   list_mod(list7)
6   list7 # observe that the changes within the function appears outside the
    ↪  function
```

Sometimes we do not know in advance how many positional or keyword arguments should be passed to the function. In these cases we can use * (or ** ) before a paramater_name in the function header to make the paramater_name a tuple (or dictionary) that can store an arbitrary number of positional (or keyword) arguments. As an example, we define a function that receives the amount of money we can spend for grocery and the name of items we need to buy. The function then prints the amount of money with a message as well as a capitalized acronym (to remember items when we go shopping!) made out of items in the grocery list. As we do not know in advance how many items we need to buy, the function should work with an arbitrary number of items in the grocery list. For this purpose, parameter accepting arbitrary number of arguments should appear last in the function definition:

```
1   def grocery_to_do(money, *grocery_items): # the use of * before grocery_items is
    ↪  to allow an arbitrary number of arguments
2       acronym = ''
3       for i in grocery_items:
4           acronym += i[0].title()
5           print("You have {}$".format(money))
6           print("Your acronym is", acronym)
7
8   grocery_to_do(40, "milk", "bread", "meat", "tomato")
```

### 1.9.2 1.8.2 Modules and Packages

As we develop our programs, it is more convenient to put functions into a separate file called module and then import them when they are needed. Importing a module within a code makes the content of the module available in that program. This practice makes it easier to maintain and share programs. Although here we are associating modules with definition of functions (because we just discussed functions), modules can also store multiple classes and variables. To create a module, we can put the definiton of functions in a file with extension .py Suppose we create a file called module_name.py and add definition of functions into that file (for example, function_name1, function_name2, . . . ). Now to make functions available in our programs that are in the same folder as the module (other- wise, the module should be part of the PYTHONPATH), we can import this entire module as:

```
1   import module_name
```

If we use this way to load an entire module, then any function within the module will be available through the program as:

```
1   module_name.function_name()
```

However, to avoid writing the name of the module each time we want to use the

function, we can ask Python to import a function (or multiple functions) directly. In this approach, the syntax is:

```
from module_name import function_name1, function_name2, ...
```

### 1.9.3 1.8.3 Aliases

Sometimes we give a nickname to a function as soon as we import it and use this nickname throughout our program. One reason for doing this is convenience! For example, why should we refer to someone named Edward as Ed? Because it is easier. Another reason is to prevent conflicts with some other functions with the same name in our codes. A given nickname is known as alias and we can use the keyword as for this purpose.

## 1.10 1.9. Iterator, Generator Function, and Generator Expression

### 1.10.1 1.9.1 Iterator

As mentioned before, within a for loop Python creates an *iterator* object from an iterable such as list or set. In simple terms, an *iterator* provides the required functionality needed by the loop (in general by the iteration protocol). But a few questions that we need to answer are the following:

- How can we produce an iterator from an iterable?
- What is the required functionality in an iteration that the iterator provides?

Creating an iterator from an iterable object is quite straightforward. It can be done by passing the iterable as the argument of the built-in function iter(): iter(iterable). An iterator object itself represents a stream of data and pro- vides the access to the next object in this stream. This is doable because this special object has a specific method __next__() that retrieves the next item in this stream (this is also doable by passing an iterator object to the built-in function next(), which actually calls the __next__() method). Once there is no more data in the stream, the __next__() raises StopIteration exception, which means the iterator is exhausted and no further item is produced by the iterator (and any further call to __next__() will raise StopIteration exception). Let us examine these concepts starting with an iterable (here a list):

```
list_a = ["a", "b", "c", "d"]
iter_a = iter(list_a)
iter_a
```

```
next(iter_a) # here we access the first element by passing the iterator to the
↪  next() function for the first time (similar to iter_a.__next__())
```

```
iter_a.__next__() # here we access the next element using __next__() method
```

```
next(iter_a)
```

```
next(iter_a)
```

Now that the iterator is exhausted, one more call raises StopIteration exception:

```
next(iter_a)
```

### 1.10.2 1.9.2 Generator Function

Generator functions are special functions in Python that simplify writing custom iterators. A generator function returns an iterator, which can be used to generate stream of data on the fly. Let us clarify this statement by few examples.

**Example 1.1** In this example, we write a function that can generate the sum of the first n integers and then use the return list from this function in another for loop to do something for each of the elements of this list (here we simply print the element but of course could be a more complex task):

```
def first_n_sum_func(n):
    sum_list_num, num = [], 1 # create an empty list to hold values of sums
    sum_num = sum(sum_list_num)
    while num <= n:
        sum_num += num
        sum_list_num.append(sum_num)
        num += 1
    return sum_list_num

for i in first_n_sum_func(10):
    print(i, end = " ") # end parameter is used to replace the default newline
    ↪ character at the end
```

In Example 1.1 we created a list within the function to hold the sums, returned this iterable, and used it in a for loop to iterate over its elements for printing. Note that if n becomes very large, we need to store all the sums in the memory (i.e., all the elements of the list). But what if we just need these elements to be used once? For this purpose, it is much more elegant to use generators. To create a generator, we can replace return with yield keyword:

```
            def first_n_sum_gen(n):
                sum_num , num = 0, 1
                while num <= n:
                    sum_num += num
                    num += 1
                    yield  sum_num

            for i in first_n_sum_gen(10):
                print(i, end = " ")

```

To understand how this generator function works, we need to understand several points:

- **item 1:** Any function with one or more yield statement(s) is a generator function;

- **item 2:** Once a generator function is called, the execution does not start and it only returns a generator object, which is an iterator (therefore, supports the __next__() methods);
- **item 3:** The first time the __next__() method (equivalently, the next() function) is called on the generator object, the function will start execution until it reaches the first yield statement. At this point the yielded value is returned, all local variables in the function are stored, and the control is passed to the calling environment;
- **item 4:** Once the __next__() method is called again on the generator, the function executation resumes where it left off until it reaches the yield again, and this process continues;
- **item 5:** Once the function terminates, StopIteration is raised

To understand the above examples, we look at another example to examine the effect of these points, and then go back to Example 1.1.

If we have an iterable and we would like to print out its elements, one way is of course what was done before in Example 1.1 to use a for loop. However, as discussed in Section 1.6.5, an easier approach is to use * as the iterable unpacking operator in the print function call (and print is a function that is defined in a way to handle that):

```
print(*first_n_sum_gen(10))
```

```
print(*first_n_sum_func(10))
```

**Example 1.2** Here we create the following generator function and call next() function several times:

```
def test_gen():
    i = 0
    print("Part  A")
    yield i

    i += 2
    print("Part B")
    yield i

    i += 2
    print("Part C")
    yield i

gen_obj = test_gen() # gen_obj is a generator object
gen_obj
```

Now we call the next() method serval times:

```
next(gen_obj)
```

```
next(gen_obj)
```

```
next(gen_obj)
```

```
next(gen_obj)
```

**Example 1.1 (continued):** Now here we are in the position to understand everything in first_n_sum_gen generator function.

Based on what we mentioned before in Section 1.9.1, in a for loop the __next__() is applied indefinitely to the iterator until it is exhausted (i.e., a StopIteration exception is raised). Now based on the aforementioned item 2, once the generator function first_n_sum_gen(10) is called, the execution does not start and it only returns a generator object, which is an iterator. The for loop applies iter() to create an iterator but applying this function to an iterator returns the iterator itself. Then applying the __next__() method by the for loop for the first time executes this function until it reaches the yield statement (item 3). At this point, sum_num and num are 1 and 2, respectively, and they are preserved and, at the same time, the value of the sum_num is returned (i.e., 1) to the caller so i in the for loop becomes 1, and is printed out. The for loop applies the __next__() method again (item 4), and the function execution resumes where it left off; therefore, both sum_num and num become 3, and the value of sum_num is returned (i.e., 3), and this continues. As we can see, in the generator function first_n_sum_gen(10), except for the local variables in each iteration, we do not need to store a possibly long list similar to first_n_sum_func. This is what really means by generating data stream on the fly: we do not need to wait until all values are generated and, therefore, less memory is used.

### 1.10.3 1.9.3 Generator Expression

The same way listcomps are useful for creating simple lists, generator expressions (known as genexps) are a handy way of creating simple generators. The general syntax for genexps is similar to listcomps except that the square brackets [ ] are replaced with parentheses ( ):

```
(expression for exp_1 in seq_1
 if condition_1
 for exp_2 in seq_2
 if condition_2
 ...
 for exp_n in seq_n if condition_n)
```

Naturally we use generator functions to create more complicated generators or to reuse a generator in different places, while genexps are used to create simpler generators and those that are not used frequently. Below is the genexp for implementing first_n_sum_gen(n):

```
n = 10
gen1 = (sum(range(1, num + 1)) for num in range(1, n + 1))
print(*gen1)
```