Bootcamp: Python - Module 2

John Rajadayakaran Edison, Sara Miner More,
Eli Sherman, Musad Haque, Soumyajit Ray

January 2023

Contents

1 Advanced Data Structures

We begin by looking at three advanced data structures for storing data available in Python: tuples, dictionaries, and sets. In the last module, we learned about lists in detail. There are many common operations and methods supported by lists, and by the data structures that we will study in this module. We will briefly mention the commonalities, and spend the rest of the time highlighting the unique features of each data structure.

1.1 Tuples

A tuple is an *immutable* collection of objects: once created, a tuple *cannot* be extended or modified. There is no restriction on the number, or type of objects, that are contained in a tuple. It is initialized by enclosing objects within parentheses

```
employee_details = ("David Brent", 42, "Slough", "Manager")
```

1.1.1 Similarities with a list

- Tuple elements are accessed in the same manner as in lists: using an index within square brackets. In the example above, `employee_details[1]` is equal to 42.
- Like lists, tuples support slicing.
- Tuples can be used as arguments to built-in functions like `len`, `min`, and `max`.
- The method `count` can be used to access the number of times an object is present in a tuple.
- The method `index` can be used to find the position of an object in the tuple.

Try it yourself

Can you predict the output of the operations listed in the code snippet below? Check your results in the interpreter.

```
# Tuples with only a single element require a comma at the end.
position_1 = (0, )*2

position_2 = position_1 + (1, 1)

position_2 += (1, 2)

print(position_2.count(2))
```

1.1.2 Unique features

- Tuples are typically used when both the order of the items in the structure and the number of items in a data structure need to remain fixed.
- Once created, the individual elements of a tuple cannot be edited, and you cannot add elements to it or remove elements from it. Line number 2 in the code snippet below will result in an error.

```
1 position_1 = (1, 0, 1)
2
3 # Tuples are immutable, so the line below will cause an error
4 position_1[1] = 2
```

- Individual elements of a tuple, once created, cannot be edited. However just as we've seen with other immutable datatypes (integers and strings), it is possible to assign a new tuple to an existing variable. The two lines of code below demonstrate this.

```
1 position_1 = (1,1,1)
2 # Reassign value
3 position_1 = (0,0,0)
```

- Since tuples have a fixed size, there are performance gains associated with creating them and reading the data in them, in comparison to lists.

1.1.3 Tuple unpacking

The elements of a tuple can be unpacked, or stored into several variables. Here is how

```
1 employee_details = ("David Brent", 42, "Slough", "Manager")
2 name, age, *other = employee_details
3 print(name, age, other)
```

```
David Brent 42 ['Slough', 'Manager']
```

The first two elements of the tuple `employee_details`, are stored in variables `name` and `age`, and the remaining elements are packaged as a list and stored in the variable `other`.

Try it yourself

What do you expect will happen upon executing the code snippet below?

```
employee_details = ("David Brent", 42, "Slough", "Manager")
*personal_info, location, title = employee_details
print(personal_info, location, title)

```

1.1.4 Are tuples strictly immutable?

A tuple can be built from objects of any datatype, including mutable datatypes like lists. Here is an example

```
name = ["David", "Brent"]
position = ["Manager", "Slough"]
salary_benefits = [45000, 3200]

employee_info = (name, position, salary_benefits)

name.append("The Rockstar")
print(employee_info)
```

The output we get is shown below:

```
(['David', 'Brent', 'The Rockstar'], ['Manager', 'Slough'], [45000, 3200])
```

Since `employee_info` is a tuple, you might expect that it cannot be changed in any manner. However, here immutability only implies that `employee_info`, will always remain a tuple of the lists `name`, `position` and `salary_benefits`. It does not place any restrictions on these lists, which can be edited and extended. In the example above, in line 7, one of lists in the tuple is being extended. Such operations are permitted and do not cause an error.

1.2 Dictionaries

A dictionary is a mutable collection of key-value pairs. It is initialized by enclosing these key-value pairs within curly braces.

In the example below, the first item in the dictionary named `top_quarterbacks` has a key `Patriots`, associated with a value `Brady`. A key in a dictionary plays the same role as an index for a list. An element of a dictionary can be accessed only with its key, not by position number.

```
# Empty Dictionary
empty_dict = {}
```

```
3
4  # A dictionary with 3 items
5  top_quarterbacks = {'Patriots':'Brady', 'Saints': 'Brees',
6  'Packers': 'Rodgers'}
7  print(top_quarterbacks)
8
9  # Display the value associated with key 'Patriots'
10 print(top_quarterbacks['Patriots'])
11
12 # Casting key value pairs into a dictionary
13 afc_coaches = dict(Patriots='Belichick', Saints='Payton')
14 print(afc_coaches)
```

```
{'Patriots': 'Belichick', 'Saints': 'Payton'}
Brady
{'Patriots': 'Brady', 'Saints': 'Brees', 'Packers': 'Rodgers'}
```

The keys in a dictionary must be unique. In order to add an item to an existing dictionary, simply use `dict_name[key] = value`. If the key already exists in the dictionary, its value will be overwritten. Otherwise, a new key-value pair is added to the dictionary.

```
1  afc_qbs = {'Patriots':'Brady', 'Jets':'Darnold','Dolphins':'Tua'}
2
3  # Edit an existing item
4  afc_qbs['Patriots'] = 'Newton'
5
6  # Add a new item
7  afc_qbs['Bills'] = 'Allen'
8
9  print(afc_qbs)
10 print(len(afc_qbs))
```

```
{'Patriots': 'Newton', 'Jets': 'Darnold', 'Dolphins': 'Tua', 'Bills': 'Allen'}
4
```

1.2.1 Similarities with lists

- Dictionaries are mutable like lists. They can be expanded and reduced in size.
- Since a dictionary is a mutable object, it should be copied via the `copy` method. Mere assignment using the = operator simply copies the reference and not the data itself.

- The size of a dictionary is given by the built-in function `len`.
- Dictionaries permit nesting, i.e., it is possible to create a dictionary of dictionaries.
- Several method that are available for lists, like `pop` for removal of a specified item and `copy` for duplication, can be employed for similar purposes with dictionaries.

Here are some additional useful dictionary methods.

`dictionary` method name	description
`items()`	Returns a list of tuples. Each tuple is a key-value pair present in the dictionary
`values()`	Returns a list of values from the dictionary
`keys()`	Returns a list of keys from the dictionary
`clear()`	Remove all key-value pairs from the dictionary
`get(key,not_present)`	This method takes two arguments. The first is a key. If the key is present in the dictionary, `get` returns the associated value. Otherwise `get` returns its second argument, which can be thought of as a default value. In the example above, try out: `afc_qbs.get('Ravens', 'Nobody')`

1.2.2 Keys and Values

- Dictionary keys are unique; the same key value cannot occur twice in the same dictionary.
- Any immutable datatype such as integer, string, or tuple can be used as the type of a dictionary key. The same type need not be used for all keys in the same dictionary.
- When a dictionary is created, its key values are hashed. In other words, a unique ID is generated for each key. Hashing enables faster membership testing, via the `in` keyword. Therefore, if in your program a lot of search operations will be performed on data, it is best to store the data in a dictionary (or a set), rather than a list.

1.2.3 Dictionary comprehensions

Dictionary comprehensions, like list comprehensions, are powerful, one-line loops that can be used for data structure creation. Here is an example in which we create several dictionaries using dictionary comprehensions.

```
1 square = {x : x**2 for x in range(1,10)}
2 print(square)
3 square_evens = {x : x**2 for x in range(2,9,2)}
```

```
print(square_evens)
another_way = {x : x**2 for x in range(1,10) if x%2 == 0}
print(another_way)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
{2: 4, 4: 16, 6: 36, 8: 64}
{2: 4, 4: 16, 6: 36, 8: 64}
```

1.3 Sets

1.3.1 Initializing sets

The `set` datatype is meant for storing a a collection of objects which disallows duplicates. Here is how you create a set in Python.

```
# Empty set
empty_set = set()

items_in_bag = set(['phone','laptop','notebook','pen'])

```

The `set()` function can take only one argument, which is a sequence (or other iterable). The following two lines are incorrect ways of initializing a set.

```
# This doesn't work
items_in_bag = set('phone','laptop','notebook','pen')

# This doesn't work either
id_numbers = set(1230,3597,4560)
```

Here's a better way to get sets created:

```
# Turn into a list before calling set
items_in_bag = set(['phone','laptop','notebook','pen'])
id_numbers = set([1230,3597,4560])
```

Members of a Python set must be hashable, i.e., they must allow conversion to a unique ID via a hash function, and must remain unchanged during the course of the program. All immutable datatypes (such as `int`, `float`, `str`, `tuple` etc.) are hashable, and can be added to a set.

The function `set()` can be used to cast a sequence, like a list or a tuple into a set. Notice below, how the set includes only the unique elements from the list.

```
# list of division-by-three remainders of first 10 numbers
remainder_3 = [x%3 for x in range(11)]
print(remainder_3)

# Turn the list into a set
set_of_remainders = set(remainder_3)
print(set_of_remainders)
```

```
[0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1]
{0, 1, 2}
```

1.3.2 A few useful methods

The objects in a set are unordered, and cannot be accessed via an index value. However, membership testing via `in` operator is possible. Here are some methods available to extend and modify a set.

`set` method name	description
`add(item)`	`add` appends the specified *item* to the set, if it is not already present in the set.
`update(seq)`	`update` takes a sequence (a list or tuple or any iterable) and adds every unique element present in the argument to the set. `update` can also handle more than one sequence at a time
`remove(item)`	Remove the element *item* from the set. If this element is not present in the set, an error is raised
`discard(item)`	Remove the element *item*, if present in the set. If not, no error is raised
`clear()`	Remove all the elements of the set

1.3.3 Similarities to lists and dictionaries

- Sets can be used as arguments to built-in functions like `len`, `min`, `max`, and `sum`. (For `min` and `max`, the elements of the set must support comparison operations. For `sum` the elements must support addition.)
- Since a set is a mutable object, it should be copied via the `copy` method. Mere assignment using the = operator simply copies the reference and not the data itself.
- The usage of the following methods is similar in sets, as in lists and dictionaries; `add`, `remove`, `pop`, `clear`, `discard`, `copy`.

- Python allows you to write set comprehensions to generate sets. They are similar to list comprehensions, except that the expression that generates the set is written within curly braces. Here is an example.

```
set_of_remainders = { n%13 for n in range(100) if n%11==0}
print(set_of_remainders)
```

```
{0, 1, 3, 5, 7, 8, 9, 10, 11, 12}
```

1.3.4 Methods unique to sets

Common set operations include taking the union of two sets, or the intersection of two sets. As shown below, union can be accomplished using the `|` operator or the `union` method; intersection uses the `&` operator or the `intersection` method.

```
menu_cafe_artifact = set(['espresso','cappucino','affogato'])
menu_cafe_wmill    = set(['espresso','shakkerato','latte'])

# Here are two ways of finding the union of two sets
all_items = menu_cafe_artifact | menu_cafe_wmill
all_items = menu_cafe_artifact.union(menu_cafe_wmill)
print("List of drinks: ",all_items)

# Here are two ways of finding the intersection of two sets
common_items = menu_cafe_artifact & menu_cafe_wmill
common_items = menu_cafe_artifact.intersection(menu_cafe_wmill)
print("Available in both: ", common_items)

```

```
List of drinks:  {'latte', 'espresso', 'shakkerato', 'cappucino', 'affogato'}
Available in both:  {'espresso'}
```

Union, intersection, difference and symmetric difference operations have a method and a corresponding operator version. In addition, there are update versions of each of these operations, which when called, overwrites the set with the result of the operation. Here is a quick review of some of these methods.

set method call example	Equivalent	Description
s.union(t)	s | t	return a set containing all unique elements in sets s and t
s.intersection(t)	s & t	return a set containing elements common to sets s and t
s.intersection_update(t)	s &= t	update set s as the set with only elements common to sets s and t
s.difference(t)	s - t	return a set with elements in set s but not in set t.
s.issubset(t)	s <= t	check if every element in set s is in t
s.issuperset(t)	s >= t	check if every element in set t is in s
s.disjoint(t)		True if there are no common elements between sets s and t

A complete list of set-related methods can be found at https://docs.python.org/3.8/library/stdtypes.html - scroll or click down to the "Set Types - set, frozenset" section.

2 Functions

As the size and complexity of the problem that you are trying to solve grows, it is beneficial to split the task into a set of small manageable tasks. This applies to Python scripts as well. Splitting the code up into smaller chunks makes it more readable and easy to maintain. Furthermore, there are cases in which a particular set of operations is repeated at various stages of the program. Typically, repeating (copying and pasting) a block of code is considered bad coding practice. Instead you could write this block once, test it thoroughly and then plug it into any of your coding projects as needed. In essence it is good practice to design your programs in a modular fashion.

Below we will learn how Python functions serve this purpose. A function is a block of code ideally designed for a specific job that gets executed only when called upon. We will learn how to (i) define and call functions (ii) package them into a module that can be plugged into any source code (iii) use anonymous one line functions or lambda expressions, and a few more interesting features. We begin by looking at an example.

2.1 A simple example

Here is a typical working day of graduate student Payd Soles. Every day at the crack of dawn he performs a series of experiments. He analyzes the measured data and fits it to the non-linear equation

$$f(x) = \exp(-ax) - bx$$

He then writes a Python script to solve this equation using the Newton-Raphson method. The details of the technique don't matter for the purposes of this document, but we've

included it just for completeness. For a non-linear equation $f(x) = 0$, the following iterative scheme yields the solution.

1. Begin with an initial guess x_0
2. Use the following equation to obtain a new estimate for x:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{1}$$

   Here i indicates the iteration number, and $f(x_i)$ and $f'(x_i)$ are evaluations of the function and its derivative, respectively, at $x = x_i$.

3. Repeat the above step until $x_{i+1} - x_i < 10^{-8}$ (tolerance)

```
# Need some math functions for function evaluation
import math

# Parameters from expt
param_a = 4.0
param_b = 2.0

# Initial guess
x_old = 0.4

# Parameters of the NR solver
tolerance = 1.0e-8
maxiter   = 1000000

# Iterative solution
for i in range(maxiter):

    f_at_x       = math.exp(-x_old*param_a) - (param_b*x_old)
    f_prime_at_x = -param_a*math.exp(-x_old*param_a) - param_b
    err          = f_at_x/f_prime_at_x
    x_old        -= err

    if abs(err)<tolerance:
        print("Results converged ")
        print(f"Solution = {x_old}")
        break

```

```
Results converged
Solution = 0.21315137550343138
```

For the next six weeks, he has to repeat this workflow each day; run a new experiment, obtain a non-linear equation and its parameters, and solve it using a Python script. The methodology to solve the equation remains the same. Only the equation and its parameters change.

Realizing it is a waste of time to repeat writing this code each day, and worried about the possibility of having to debug it all, he decides to write a function to solve non-linear equations. Once tested, this function can be simply called from any script to achieve the same result as the piece of code above. Here is how you can do the same, using functions in Python.

2.2 Defining a function

Here is the piece of code written above, this time within a function.

```
# Need some math functions for function evaluation
import math

# Parameters of the NR solver
tolerance = 1.0e-8
maxiter   = 1000000

# Define function to solve equation
def solver_nr (param_a, param_b, x_old):

    # Iterative solution NR Method
    for i in range(maxiter):

        f_at_x       = math.exp(-x_old*param_a) - (param_b*x_old)
        f_prime_at_x = -param_a*math.exp(-x_old*param_a) - param_b
        err   = f_at_x/f_prime_at_x
        x_old -= err
        if abs(err)<tolerance:
            print(f"Solution = {x_old}")
            return x_old

result = solver_nr ( 4.0, 2.0, 0.2)
```

```
Solution = 0.21315137550343136
```

You might have noticed two new python keywords in the above block of code; `def` and `return`. The first is used to define a function. What follows the `def` keyword is the name of the function, in this case being `solver_nr`. Within parentheses is the information that is fed as input to the function. Information is fed via variables which are also called the

parameters of the function. There is no restriction on the number or the data type of the parameters. Line number 22 in the code snippet above, is a function call written outside the function (note the indentation level). We will learn more about function calls later in this section.

2.3 Scope of variables

The variables that are defined in the function like `err`, `f_at_x` etc are visible only within the function. In other words they exist only within the local scope of the function. The variables defined prior to the function definition like `tolerance` and `maxiter` are global variables which can be accessed within the function as well as outside of it.

As with loops and conditional statements in python, scope is determined by indentation. It is critical to be careful to use consistent indentation throughout a Python program as it is possible to have syntactically correct indentation that yields a bug (for instance if you accidentally un-indent the last line of code in a loop, conditional, or function).

2.4 `return` statement

The *indented* block of statements directly below the function definition statement (i.e. the `def myFun()` line of code) is executed when the function is called. This block typically includes an optional `return` command, a Python keyword, which terminates the execution of the function.

The `return` keyword can be invoked with zero or more expressions (including names of variables that are referenceable from within the function) following it in the same line of code. When the Python virtual machine executes the `return` statement it evaluates these zero of more expressions and returns back the appropriate value(s). If zero expressions follow – simply invoking `return` alone – the Python keyword `None` is returned back. If one or more expressions follow, they must be separated by commas, and the values of the expressions will be returned back. If a function does not have a `return` statement, `None` is returned by default.

In the above example, our `return` statement had a single expression. When `return` was read and executed by the virtual machine, the values of `x_old` was evaluated and returned back to the caller.

2.5 Calling a function

The statement in line number 22 in the previous example is called a function call; it is nothing but the name of the function followed by a list of values which are being passed to the function. These values are called *arguments* to the function. The argument values get mapped to the parameters of the function, in the order in which they were expressed in the call, i.e `param_a` takes the value 4.0, `param_b` takes the value 2.0 and `x_old` takes 0.4. The number of arguments in the call should match the number of parameters specified in

the function definition. These arguments are called *positional arguments*, as their order in the call matters.

Alternatively, instead of using positional arguments, a set of *keyword arguments* or name-value pairs can be used in a function call. When using keyword arguments, the order of the name-value pairs *does not* matter, as illustrated in the following example.

```
# Using a list of parameter = argument expressions
result = solver_nr ( param_a = 4.0, param_b = 2.0, x_old = 0.4 )

# Order of the arguments doesn't matter when parameter names used
result = solver_nr ( param_b = 2.0, x_old = 0.4, param_a = 4.0 )
```

One may also choose to use a mixture of positional and keyword arguments in a function call. However the positional arguments should be specified first, followed by the keyword arguments.

```
result = solver_nr ( 4.0, param_b = 2.0, x_old = 0.4 )
```

One could also simply pack the arguments as a tuple or a dictionary, as shown below.

```
#Tuple of arguments
arg_tuple = (4.0, 2.0, 0.4)

# Unpack the tuple to send positional arguments
result = solver_nr ( *arg_tuple )

# Dictionary of arguments
arg_dict = {"param_a":4.0, "param_b":2.0, "x_old": 0.4 }

# Unpack the dictionary to send keyword arguments
# Dictionaries are unpacked with **, not *. Keys and values are sent.
result = solver_nr ( **arg_dict )

```

2.6 Optional arguments

In the above example, the variables `maxiter` and `tolerance` are strictly related to the solver, and have no other role to play. It is best if they are moved into the function as parameters, which also provides the flexibility of modifying their values via arguments. However, this creates an extra burden for the user, as two additional arguments have to be passed each time. To avoid this, we can take advantage of providing default values for these arguments. Here is how

```
# Need some math functions for function evaluation
import math

# Define function to solve equation
def solver_nr (param_a, param_b, x_old, maxiter = 1000000,
               tolerance = 1.0e-8):

    # Iterative solution NR Method
    for i in range(maxiter):

        f_at_x       = math.exp(-x_old*param_a) - (param_b*x_old)
        f_prime_at_x = -param_a*math.exp(-x_old*param_a) - param_b
        err   = f_at_x/f_prime_at_x
        x_old -= err
        if abs(err)<tolerance:
            print(f"Solution = {x_old}")
            return x_old

result = solver_nr (4.0, 2.0, 0.4)
```

```
Solution = 0.21315137550343138
```

Since we have provided the default values for the two parameters `maxiter` and `tolerance`, they are optional arguments and can be left unmentioned during a function call. In its current form, the function `solver_nr` is not very useful, as it is written for the narrow purpose of solving one non-linear equation. Before we think of ways to improve this function, let us learn about a few more features related to functions available in Python.

2.7 Lambda expressions

A lambda expression is one-line expression, the result of which yields a *function object* which can be called for execution. Let's begin with an example.

```
double_input = lambda x : 2*x
print(type(double_input))
print(double_input(4.0))
```

```
<class 'function'>
8.0
```

The expression `lambda x : 2*x` can be read as follows; it is a function that takes one parameter `x` and returns the value `2*x`. What follows the `lambda` keyword is a list

of parameters, followed by a colon symbol and a valid Python expression. Python functions can be assigned to variables, and as you can from the code snippet above, `double_input` is an object of type function. It can also be combined with the conditional expressions that we learned in the last module. Can you guess what the lambda function written below does?

```
odd_or_even = lambda x : print("Even") if x%2==0 else print("Odd")
print(odd_or_even(5))
```

```
Odd
None
```

2.8 Functions are first-class objects

Python functions are *first-class objects*. What this means is that they can be used in ways similar to other objects like integers or strings, in that: (i) they can be stored in variables (ii) they can be stored in data structures like lists or tuples or dictionaries (iii) they can be passed as arguments to a function and (iv) they can also be returned by a function. Let's see a small demonstration.

```
def get_func_raised_to_n(n):

    # Define a function based on n
    def my_func(x):
        return x**n

    # Return the function defined above
    return my_func

# Storing a function in a variable
raised_to_two  = get_func_raised_to_n(2)

print(type(raised_to_two))
print(raised_to_two(9))
```

```
<class 'function'>
81
```

The function `get_func_raised_to_n` when called, returns a function `my_func` which is defined based on the parameter `n`. Since `my_func` is defined within another function, it is only visible within this local scope. In line number 11 the function call

returns a *function* which squares the input. This function is then stored in the variable `raised_to_two`. The two `print` statements in line numbers 13 and 14 show that this variable has the `type` of a function object and that it can be called by passing arguments. Another way to define the `get_func_raised_to_n` function in the style of Python would be:

```
1 def get_func_raised_to_n(n):
2     # Return a function that raises x to power of n
3     return lambda x: x**n
```

2.9 Improvements to the solver

After having learned these new features, here are a few tweaks which will improve the quality of the Newton Raphson solver function written above.

- Generalize the solver function to any non-linear equation. This can be done by passing in as arguments two functions, which return the value of the equation and its derivative, respectively.
- Return a flag that confirms that convergence has been achieved.
- Add documentation in the form of a docstring to the function (see below).
- Package the solver into a module.

```
1  import math
2
3  # Define function to solve equation
4  def solver_nr (f_eval, f_prime_eval, x_old, maxiter = 10000,
5                 tolerance = 1.0e-8):
6      """
7      Function : solver_nr : Newton Raphson Solver
8      Arguments :
9
10     f_eval : Evaluate equation to be solved (function)
11     f_prime_eval : Evaluate first derivative of
12     equations to be solved (function)
13     x_old : Initial guess
14
15     Optional Arguments:
16
17     maxiter : Maximum number of iterations (10000)
18     tolerance : Convergence tolerance of solver (1.0e-8)
19
```

```
    Returns :
    is_converged : Boolean for deciding if iterations converged
    x_new : Solution of the non-linear equation
    """
    # Iterative solution NR Method
    for i in range(maxiter):
        err    = f_eval(x_old)/f_prime_eval(x_old)
        x_old -=  err

        if abs(err)<tolerance:
            print(f"Solution = {x_old}")
            return True, x_old

    else:
        return False, -1.0

if __name__ == "__main__":
    param_a      = 4.0
    param_b      = 2.0
    f_of_x       = lambda x : math.exp(-x*param_a) - (param_b*x)
    f_prime_of_x = lambda x : -param_a*math.exp(-x*param_a) - param_b
    is_converged, result = solver_nr ( f_of_x, f_prime_of_x, 0.4)
```

```
Solution = 0.21315137550343138
```

2.10 Packaging the function into a module

Let's save the above script as `solvers.py`. If this script is run using the command `python solvers.py`, then the indented block within the `if` statement is executed. The `if __name__ == "__main__":` statement simply checks if this script is being loaded as the main module, or if it is being imported by another script. You have now packaged your function in a module. You could consider adding more solvers as functions to this module; e.g. Secant method, fixed point iteration scheme, etc.

You may have guessed how the solver function written above can be imported into another script. We did this exact thing when we used the `exp` function from the `math` module. Here is a short script that imports `solvers.py` to access the function written in it.

```
import solvers
param_a      = 4.0
param_b      = 2.0
f_of_x       = lambda x : math.exp(-x*param_a) - (param_b*x)
f_prime_of_x = lambda x : -param_a*math.exp(-x*param_a) - param_b
is_converged, result = solvers.solver_nr (f_of_x, f_prime_of_x, 0.4)
```

The function call (line 6) has to be in the form i.e. module name dot function name, i.e., `solvers.solver_nr ( ... )`. It is also possible to import a specific function using `from solvers import solver_nr`. If imported in this fashion, the function call is simply `solver_nr (f_of_x, f_prime_of_x, 0.4)`. It is also possible to import a module or a function from a module and provide it an alias using the `as` keyword.

```
1 import solvers as slv
2 from solvers import solver_nr as nr_slv
```

The corresponding functions calls would then be

`slv.solver_nr (f_of_x, f_prime_of_x, 0.4)` and

`nr_slv (f_of_x, f_prime_of_x, 0.4)` respectively.

Now you might wonder how Python knows where your file `solvers.py` is located. Python searches a select set of folders for the presence of a module. These locations can be viewed and modified by accessing the list named `path` stored in the module `sys`. The list is initialized based on the `PYTHONPATH` environment variable.

```
1 import sys
2 print(sys.path)
3
4 # Add the location where solvers.py is located
5 sys.path.append('/home/dayakaran/my_solvers')
```

```
['', '/Applications/Anaconda/anaconda3/lib/python3.7/site-packages', '/Library/Frameworks/
Python.framework/Versions/3.6/lib/python36.zip', '/Library/Frameworks/Python.framework/
Versions/3.6/lib/python3.6', '/Library/Frameworks/Python.framework/Versions/
3.6/lib/python3.6/lib-dynload', '/Library/Frameworks/Python.framework/Versions/3.6/
lib/python3.6/site-packages']
```

2.11 Docstrings

The piece of text between the triple quotes, below the function definition, is called a docstring. It is a multi-line comment, that can be used to provide useful information about the function. It is considered good programming practice to include a docstring with every function you write, as it improves the readability of the code. Users who are unaware of the inner workings of your code can benefit from docstrings. They can use the built-in function `help()` to read your docstring. e.g `help(solver_nr)`.

3 A few extra notes on functions

3.1 Generators and `yield`

Python functions can also be used to create generators. What is a generator? Let us assume you'd like to iterate over the first billion numbers in the Fibonacci sequence. One option, is to write a function that returns a list of these numbers. However, this returned list will consume a lot of space in memory. Instead, it would be ideal to have a function which provides just one number of the sequence at a time, in an "on-demand" fashion. Such a function is called a generator.

```
1  # Generator Fibonacci sequence upto n terms
2  def fibo(nterms):
3      first, second = 0, 1
4      count = 0
5      while count < nterms:
6          yield first
7          first, second = second, first + second
8          count += 1
9
10 # Store generator in variable
11 fib_seq = fibo(15)
12 print("Type of fib_seq is :", type(fib_seq))
13
14 # Iterate using the generator
15 for i in fib_seq:
16     print(i, end=" ")
```

```
Type of fib_seq is : <class 'generator'>
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

In the above piece of code the variable `fib_seq` is a generator of values which can be iterated over. During the first iteration of the loop (line 14) the program executes the list of statements in the function until the `yield` statement is reached. `yield` is similar to `return` in that, it can send back values to the function call. Upon execution of `yield` the function ceases temporarily, however, the variables defined within the function, `count, first, second..` retain their values in memory. The execution then moves to the body of the `for` loop, and during the subsequent iteration, the execution moves back to the statement following the `yield` statement within the function (line 7). This keeps repeating until the `while` loop in the function is exhausted. In short, `fibo` generates one value of the sequence at a time, on demand.

3.2 One-line generators

It is also possible to write one-line versions of generators. Here is a one-line generator, that gives the square of the first `n` numbers in a list. It resembles a list comprehension, with the difference being that the expression is enclosed in parentheses. Notice the difference in memory consumed by the generator vs the list.

```
n = 1000000

# List of square of n - numbers
square_list = [x*x for x in range(n)]
# Generator of square of n - numbers
square_gen = (x*x for x in range(n))

import sys
# Find memory consumption
print(f"Memory used by list: {sys.getsizeof(square_list)} bytes")
print(f"Memory used by generator: {sys.getsizeof(square_gen)} bytes ")

```

```
Memory used by list: 8697464 bytes
Memory used by generator: 88 bytes
```

3.3 Args And kwargs

Python allows the user to send an arbitrary number of keyword based or positional arguments to a function. Here are two examples that demonstrate this.

```
# Parameter is a packaged tuple all_nums
def add_numbers (*all_nums):
    sum_nums = 0
    for num in all_nums:
        sum_nums += num
    return sum_nums

print(add_numbers(1,45,7))
print(add_numbers(1,4,17,96))
```

```
53
118
```

```
# Receive a number of keyword arguments
# packaged as a dictionary
def print_dict(**my_dict):
    for key, value in my_dict.items():
        print(f"Team :{key}, QB: {value}")

# The key value pairs below are packaged into a dict automatically
print_dict (patriots = 'Brady', saints = 'Brees',packers = 'Rodgers')
```

```
Team :patriots, QB: Brady
Team :saints, QB: Brees
Team :packers, QB: Rodgers
```

3.4 What happens when a mutable argument is passed

All the examples we saw so far used immutable datatypes as arguments. Here is something to keep in mind, when using a mutable datatype like a list as an argument. When the interpreter reads a function definition, it evaluates the default arguments once. In the example shown below, the default argument is an empty list. The interpreter creates an empty list object in memory and attaches the label `my_list` to it. Every call to the function that does not override the default value of the argument continues to modify `my_list`.

```
def make_list(x,my_list=[]):
    my_list.append(x)
    return my_list

print(make_list(1))
print(make_list(2))
print(make_list(3))
```

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

The function call in line 6 (and 7) `make_list(2)` does not imply `make_list(2, [])`. It simply says, modify the existing list (`my_list`), that was created when the function was first defined.

Try it yourself

Can you think of a way to resolve the above issue? Can you rewrite the function such that lines 6 and 7 in the above code will return `[2]` and `[3]` respectively?