COURSE BOOK

Programming with Python

DLMDSPWP01

iubh INTERNATIONALE HOCHSCHULE

Course Book

Programming with Python

DLMDSPWP01

iubh INTERNATIONALE HOCHSCHULE

Masthead

Publisher:
IUBH Internationale Hochschule GmbH
IUBH International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing address:
Albert-Proeller-Straße 15-19
D-86675 Buchdorf

media@iubh.de
www.iubh.de

DLMDSPWP01
Version No.: 001-2020-0728

Module Director

Dr. Reza Shahbazfar

Mr. Reza Shahbazfar teaches in the field of data science and artificial intelligence in IUBH's distance learning programs. He focuses on the fields of Programming.

Mr. Shahbazfar studied mechanical engineering as well as renewable energies and energy efficiency at the University of Kassel and received his doctorate at the Technical University Darmstadt. Some of his previous professional positions include working as a developer, project manager, researcher, and consultant in numerous projects. The focus of his work is on the development and application of modern methods of computer science.

He has been a full stack developer for several years and has been using Python and modern data science methods in numerous research and digitization projects in industry.

Table of Contents

Programming with Python

Introduction

Programming with Python

iubh INTERNATIONALE HOCHSCHULE

Signposts Throughout the Course Book

Welcome

This course book contains the core content for this course. Additional learning materials can be found on the learning platform, but this course book should form the basis for your learning.

The content of this course book is divided into units, which are divided further into sections. Each section contains only one new key concept to allow you to quickly and efficiently add new learning material to your existing knowledge.

At the end of each section of the digital course book, you will find self-check questions. These questions are designed to help you check whether you have understood the concepts in each section.

For all modules with a final exam, you must complete the knowledge tests on the learning platform. You will pass the knowledge test for each unit when you answer at least 80% of the questions correctly.

When you have passed the knowledge tests for all the units, the course is considered finished and you will be able to register for the final assessment. Please ensure that you complete the evaluation prior to registering for the assessment.

Good luck!

Learning Objectives

The course **Programming with Python** aims to lay a solid foundation in order to master the Python programming language.

Python has consistenly been one of the most popular programming languages in the last few years. As of June 2020, it holds the number one position on PYPL, which is the Popularity of Programming Language Index. This popularity stems from a clean and concise language design, Python's support for multiple programming paradigms, and a rich ecosystem of libraries that makes the language suitable for a wide range of application domains. For data science and artificual intelligence in particular, Python and R are the most widely used languages today.

In this course, you will first be introduced to the fundamentals of the language, for example, basic data structures, functions, flow-control, input-output, and the structure of Python modules and packages. Building on that foundation, the most salient concepts of object-oriented programming (OOP) are outlined and important issues concerning error handling in Python are explained. Working with the most important libraries for scientific computing, data science, and artificial intelligence is an important part of the work of any data scientist, so one unit is solely devoted to this highly relevant subject. The course closes with an overview of significant aspects of applied programming such as Python environments and version control.

Unit 1

Introduction to Python

STUDY GOALS

On completion of this unit, you will have learned...

- ... the different data structures that are used in Python.
- ... how to design and implement functions.
- ... how to control data flow and create loops.
- ... the different types of data input and output.
- ... how to create modules and packages.
- ... how to organize programming code using modules and packages.

1. Introduction to Python

Introduction

Python is the most popular open source programming language today. It is a high-level, interpreted, general-purpose programming language that was created by Guido van Rossum and first released in 1991. Python can be used to develop desktop GUI applications, websites, web applications, complex scientific and numeric programs, etc. It is free, portable, powerful, and remarkably easy to use. Below are some of the main reasons for Python's popularity:

- easy to code and to maintain,
- an object-oriented programming (OOP) language,
- compatible with Windows, Linus, Apple iOS, and Google OS operating systems,
- robust core standard library,
- many open source frameworks and libraries are part of the Python Ecosystem,
- simple general-purpose programming language, and
- unit test implementation.

This unit will primarily examine four types of data structure used in Python: integers, floats, strings, and Booleans. In relation to this, non-primitive data structures and unordered object sequences will also be covered.

Functions as reusable pieces of code are the main units of the procedural programming paradigm — we will examine them in this unit along with conditional flow control, data input, and data output.

Python programming code for application development is organized using modules and packages. The final section in this unit will discuss different aspects of modules and packages and explain how to implement them.

1.1 Data Structures

In computer programming, a variable is used to store and label data so that it can be referenced and manipulated (Kapil, 2019). In Python, there is no need to declare a variable at all. If you assign the value 3.14 to a pi variable, the Python interpreter assumes that pi is a float variable. Python is a dynamically typed language where, at runtime, the type of data can be stored as a mutable object, meaning that it can be changed after it has been created (Beazley & Jones, 2013).

Identifiers are used in Python to name objects. An identifier starts with a letter A (or a) to Z (or z), or an underscore (_) followed by a zero, more letters, underscores, or any digits (0 to 9). For example: `last_name, _last_name, last_name_1`, etc. It is very important to follow the Python naming conversion specified in the PEP 8 -- Style Guide

for Python Code (where PEP stands for Python Enhancement Proposal). This collection of documents describes Python standards, code organization, and development best practices.

Although there are many options when it comes to naming objects in Python, there are some words that cannot be used since they already have a specific function. The following words are reserved and cannot be used to declare any object:

Reserved Words				
False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Data structures are collections of related data that define the relationship between the data and determine operations that can be performed on the data. In computer science, data structures are divided into two main types: primitive and non-primitive.

Primitive Data Structures

Primitive data structures are the main building blocks for data manipulation in Python. The following are all primitive data structures (Beazley & Jones, 2013):

Primitive variables
These are the most basic data structures, hence their "primitive" title. They are capable of storing a single value.

1. Integers are numerical data including whole numbers from —∞ to +∞, for example, —1, 0, and 1.
2. Floats are floating point numbers used for rational numbers with a decimal point, for example, —1.1, 1.1, and 2.2.
3. Strings are a sequence of characters, words, digits, etc. In Python, strings are created by enclosing a sequence of characters within a pair of single or double quotation marks, for example, 'calculator' or "calculator." In this course book, we will be using double quotation marks to represent a string variable in Python.
4. Boolean represents two constant values as True and False. They are represented internally by the numbers 1 and 0. Variables of Boolean data type are used frequently in conditional and comparison expressions.

Example:

```
def main():
#  integer variable
   year = 2019
   print("The year value is: {}".format(year))

# float variable
   pi = 3.14
   print("The pi value is: {}".format(pi))

# string variable
   program_version = "Python 3.7 "
   print("The program version value is: {}".format(program_version))

# boolean variable
   file_exists = True
   print("The file exists value is: {}".format(file_exists))

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

Result:

```
The year value is: 2019

The pi value is: 3.14

The program version value is: Python 3.7

The file exists value is: True
```

Non-Primitive Data Structures

Non-primitive data structures represent ordered and unordered sequences of objects in Python. These structures, including their descriptions, are listed below:

1. Lists are ordered sequences of any objects enclosed in brackets, for example: `list_example = [1, 2, 3, "a", "b", "c"]`. These are mutable, which means that you can change their content without changing their identity.
2. Tuples are ordered sequences of any objects enclosed in parentheses, for example: `tuple_example = (1, 2, 3, "a", "b", "c")`. These are immutable, meaning that you cannot change their content without changing their identity.
3. Dictionaries are ordered sequences of any objects that are key-value pairs enclosed in curly brackets, for example: `dictionary_example = {1: "a", 2: "b", 3: "c"}`. These are mutable objects.

4. Sets are unordered sequences of unique objects, for example: `set_example = set([1, 2, 3, 4, 5])`. These are mutable objects.
5. Frozensets are immutable sets objects, for example: `frozenset_example = frozenset({1, 2, 3, 4, 5})`.

We will further examine how Python defines mutable and immutable objects. A mutable object can be changed after it is created, whereas an immutable object cannot. Built-in objects such as integer, float, Boolean, string, and tuple are immutable. Other built-in objects such as list, set, and dictionary are mutable.

Example:

```
def main():

   list_example = [1, 2, 3, "a", "b", "c"]
   print("An example of a list is: {}".format(list_example))

   tuple_example = (1, 2, 3, "a", "b", "c")
   print("An example of a tuple is: {}".format(tuple_example))

   dictionary_example = {1:"a", 2:"b", 3:"c"}
   print("An example of a dictionary is: {}".format(dictionary_example))

   set_example = set([1, 2, 3, 4, 5])
   print("An example of a set is: {}".format(set_example))

   frozenset_example = frozenset([1, 2, 3, 4, 5])
   print("An example of a frozenset is: {}".format(frozenset_example))

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

Result:

```
An example of a list is: [1, 2, 3, 'a', 'b', 'c']

An example of a tuple is: (1, 2, 3, 'a', 'b', 'c')

An example of a dictionary is: {1: 'a', 2: 'b', 3: 'c'}

An example of a set is: {1, 2, 3, 4, 5}

An example of a frozenset is: frozenset({1, 2, 3, 4, 5})
```

String Manipulation

Python string objects are immutable. The following code example explains this concept (Beazley & Jones, 2013).

Example:

```
def main():
 #    strings are immutable objects
   string_value = "python"
   print(string_value)
   print(string_value[5])

 #    an error occurred. string_value[5] can't be changed
   string_value[5] = "m"
   print(string_value[5])

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

Results:

```
python

n

Traceback (most recent call last):

File "C:\path_folder\eclipse-workspace\iubh_fernstudium\src\1.1_data_structures
\string_manipulation.py", line 60, in main

string_value[5] = "m"

TypeError: 'str' object does not support item assignment
```

As you can see, the value of the `string_value[5]` cannot be changed to “m” because it was already set to “n.”

Python contains many string manipulation methods. We will now look at the most commonly used methods.

Example:

```
def main():
#    string length using len() method
   string_0 = "Python is becoming the worlds most popular programming
    language today" print(len(string_0))
```

```
#    string concatenation using + sign
   string_1 = "Python is becoming the worlds most " + "popular programming
    language today" print(string_1)
#    string concatenation using join() method
   string_2 = "".join(("Python is becoming the worlds most " ,
    "popular programming language today"))
   print(string_2)
#    substrings (or splices) using [start, start + length]
#    get 'Python' word from the start
   string_3 = string_0[0:6]
   print(string_3)
#    remove 'Python' word from the start
   string_4 = string_0[7:]
   print(string_4)
#    remove 'today' word from the end
   string_5 = string_0[:-6]
   print(string_5)
#    select 'programming' word only
   string_6 = string_0[43:54]
   print(string_6)
#    count how many times letter 'a' is in string
   string_7 = string_0.count("a")
   print(string_7)
#    find what position 'language' word starts
   string_8 = string_0.find("language")
   print(string_8)
#    remove any whitespace from the start to end
   string_9 = string_0.strip()
   print(string_9)
#    convert to lower case
   string_10 = string_0.lower()
   print(string_10)
#    convert to upper case
   string_11 = string_0.upper()
   print(string_11)
#    replace 'Python' word with 'Java'
   string_12 = string_0.replace("Python", "Java")
   print(string_12)
#    split string into substring in 'most' word
   string_13 = string_0.split("most")
   print(string_13)
#    check if 'popular' word is in string
   string_14 = "popular" in string_0
   print(string_14)
#    check if 'POPULAR' word is in string
   string_15 = "POPULAR" not in string_0
```

```
   print(string_15)
if __name__ == '__main__':
   main()
```

Result:

```
69

Python is becoming the worlds most popular programming language today

Python is becoming the worlds most popular programming language today

Python

is becoming the worlds most popular programming language today

Python is becoming the worlds most popular programming language

programming

5

55

Python is becoming the worlds most popular programming language today

python is becoming the worlds most popular programming language today

PYTHON IS BECOMING THE WORLDS MOST POPULAR PROGRAMMING LANGUAGE
TODAY

Java is becoming the worlds most popular programming language today

['Python is becoming the worlds ', ' popular programming language today']

True

True
```

1.2 Functions

A **function** is a reusable piece of code that is written to solve a predefined task. In general, the main idea of creating functions in computer programming is to create a subprogram that can be called as many times as needed. This will avoid code duplication where possible. Functions are the main components of class objects, and the function name must follow the Python naming convention (Beazley & Jones, 2013).

Function
A function is a block of code that only runs when it is called.

Python uses the following three types of functions:

1. Built-in functions — These are already developed functions inside the Python interpreter. For example, "print()" is used to print any object to the terminal and "max()" is used to get the maximum value of any numerical variable.
2. User-defined functions — This is a custom function developed by a user for their daily program needs.
3. Anonymous functions — These are functions that are defined without a name. Anonymous functions are defined using the "lambda" keyword. In general, these functions run quickly in Python.

Examples of Functions

User-defined functions are declared using the "def" keyword. A simple example of defining a function is shown below:

```
def function_name(parameters):
    """
    function comments or docstring (function task, parameters definition, etc.)
    """
    <statement>
    ...
    return <expression>
```

Python uses the "docstring" term (the text in triple quotes) like a comment to document modules, functions, and classes. A "docstring" is a string literal that occurs as the first statement in a module, function, or class.

The main idea of the method is to encapsulate a computation together with its scope so that it can be treated as primitive (Ramalho, 2015). Parameters are the method arguments passed to the function. Sometimes, these parameters are called input parameters and they are passed by value or by reference. To pass by value means that a copy of the actual parameter's value is made in memory. When pass by reference (by address) occurs, a copy of the address of the actual parameter is stored.

By default, Python passes parameters by reference and does not support output parameters like other languages. There can be one or many parameters. The code block of the function includes the docstring and code statements (lines of code). The colon

symbol ":" after parentheses creates an indented block. Python uses indentation to make the code organized, clean, and easy to understand. In general, the amount of indentation is four spaces, but it can be any consistent number throughout that block. The "return" statement is used to return a value to the calling function. The follow function prints a passed message:

Example:

```
def print_message(message):
   print("The message is: {}".format(message))

def main():
   my_message = "I like Python programming language"
   # calling function
   print_message(my_message)

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

Result:

```
The message is: I like Python programming language
```

The `if __name__ == '__main__'`: statement checks whether the module name is equal to the `__main__` built-in variable in Python. It tells the module to start after this line. It is good programming practice to start any Python program with this standard main statement.

As you can see, this simple function does not return any value. If a function needs to return a value, the "return" statement is required. We will now examine the function "`addition_two_number`" below.

```
def addition_two_number(number_1, number_2):
   """
   calculate the addition of two numbers
   :param number_1: first number
   :param number_2: second number
   :return: sum of first number plus
   """
   number_addition = number_1 + number_2
   return(number_addition)
```

This function calculates the addition of two integer numbers. If you call this function

```
number_1 = 100
number_2 = 200
result = addition_two_number(number_1, number_2)
print("The result is: {}".format(result))
```

Result:

```
The result is: 300
```

Function Default Parameters

Default parameters take a default value using the assignment operator **=**. However, they can be overwritten if another value is passed to the function. In the example below, the "`payment_day()`" function calculates the total payment by day when the payment by hour is fixed at 25.

Example:

```
def payment_day(working_hours, payment_hour=25):
    '''
    calculate payment by day
    :param working_hours: amount of working hours
    :param payment_hour: payment by hour
    :return: payment by day
    '''
    total_payment = working_hours * payment_hour
    return total_payment
```

Calling this function:

```
working_hours = 8
# calling function
total_payment = payment_day(working_hours)
print("The total payment is: {}".format(total_payment))
```

Result:

```
The total payment is: 200
```

The payment for 8 hours a day is 200 when the payment per hour is 25. As you can see in the calling statement, the amount of payment per hour does not need to be passed.

In some cases, you do not know the default parameter value. In these cases, the "None" keyword can be used to define a null value or no value at all. It is important to mention that "None" is not the same as 0, False, Null, or an empty string. None is a datatype of its own. The function "payment_day()" can be updated as

```
def payment_day(working_hours, payment_hour=None):
    '''
    calculate payment by day
    :param working_hours: amount of working hours
    :param payment_hour: payment by hour
    :return: payment by day
    '''
    if payment_hour is not None:
        total_payment = working_hours * payment_hour
    else:
        total_payment = 0
    return total_payment
```

The payment by day is only calculated when payment by hours is not None, otherwise it will be 0. There are two ways to call this function.

Example:

```
working_hours = 8
total_payment = payment_day(working_hours)
print("The total payment is: {}".format(total_payment))
```

Result:

```
The total payment is: 0
```

Example:

```
working_hours = 8
payment_hour = 25
total_payment = payment_day(working_hours, payment_hour)
print("The total payment is: {}".format(total_payment))
```

Result:

```
The total payment is: 200
```

Having two or more functions with the same name and different input parameters is called method overloading. Instead of using many functions for the required task(s), you can encapsulate everything in one function, reducing the amount of code considerably.

Variable Number of Parameters

There may be a case when we do not know the exact amount of input parameters to be passed to a function. In a case such as this, the following syntax single-asterisk parameters *args or double-asterisk **kwargs would to be used. We will now examine the program below using the single-asterisk parameters *args.

Example:

```
def sum_number(*args):
 total_sum = sum(args)
 return total_sum

def main():
   result_1 = sum_number(1, 2, 3, 4, 5)
   print("result_1: {}".format(result_1))

   result_2 = sum_number(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
   print("result_2: {}".format(result_2))

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

Result:

```
result_1: 15

result_2: 55
```

Because *args was used to send a variable-length argument list to the `sum_number()` function, we can pass as many parameters as necessary to the calling function.

The double-asterisk parameter **kwargs can be used to pass variable-length dictionary structure to the function. The function below is an example of this.

Example:

```
def print_dictionary(**kwargs):
   for key, value in kwargs.items():
       print("The value of {}: {}".format(key, value))

def main():
   print_dictionary(value1=1, value2="Python", value3=False, value4=3.14)

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

Result:

```
The value of value1: 1

The value of value2: Python

The value of value3: False

The value of value4: 3.14
```

Like the *args, we can use **kwargs to pass many parameters to a function.

Variable Scopes

The place in the program where a name (variable) is defined is called scope. Python supports two types of variable scopes.

1. Local variables — By default, this is any variable defined inside the function.
2. Global variables — This is any variable defined outside the function. If a global keyword is used inside the function, a variable will become global for the whole program. Global variable declaration is not often used in Python today.

Example:
x_global = 100

```
def calculate_double(x):
   x_local = x * 2
   return x_local

def main():
   value = 10
   value_double = calculate_double(value)
   print("The value double in main() is: {}".format(x_global))
   #    print("The x local in main() is: {}".format(x_local))

if __name__ == '__main__':
   main()
   print("The x global after main() is: {}".format(x_global))
```

Result:

```
The value double in main() is: 100

The x global after main() is: 100
```

The `x_global` variable is defined at the top of the program, indicating that global will be used for the whole program. It can be used anywhere. If we try to print the `x_local` variable,

```
if __name__ == '__main__':
   main()
   print("The x global after main() is: {}".format(x_global))
   print("The x local after main() is: {}".format(x_local))
```

an error will occur.

```
Traceback (most recent call last):
   File "C:\folder_path\workspace\iubh_fernstudium\src\1.2_functions\function.py",
   line 144, in <module>
   print("The x local in main() is: {}".format(x_local))
NameError: name 'x_local' is not defined
```

However, if we declare the variable `x_local` as global,

```
def calculate_double(x):
   global x_local
   x_local = x * 2
   return x_local
```

the program will run without any errors at all.

```
The value double in main() is: 100
The x global after main() is: 100
The x local after main() is: 20
```

Nested Functions

A nested function is a function that is defined inside another function. The nested function can access the variables in the enclosing function as read only. To change the value of any of these variables in the nested function, they need to be declared by explicitly using the nonlocal keyword.

Example:

```
def outer_function(x):
   ''' enclosing function
   '''
   def inner_function():
       '''
       nested function
       '''
       # nonlocal x
       x = 5
```

```
        print("The x value inside inner function is: {}".format(x))
        x_local = x * 2
        print("The x local value inside inner function is
             {}".format(x_local))

    inner_function()
    print("The x value outside inner function is: {}".format(x))

def main():
    value = 10
    value_double = outer_function(value)

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

Result:

```
The x value inside inner function is: 5

The x local value inside inner function is: 10

The x value outside inner function is: 10
```

If one uncomments the line nonlocal x by removing the hash sign at the beginning of the row, the result will be:

```
The x value inside inner function is: 5
The x local value inside inner function is: 10
The x value outside inner function is: 5
```

The x value changes from 10 to 5 when the scope changes due to the use of the nonlocal statement.

Lambda Functions

Lambda functions are known as anonymous functions as they have no name. The return statement is not required as they always return a value. The best way to use these functions is inside another standard def function.

Example:

```
def main():
    function_double = lambda x : x * 2
    x = 10
    print("The x value is: {}".format(function_double(x)))
```

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

Result:

```
The x value is: 20
```

1.3 Flow Control

Conditional Statements

By definition, conditional statements are features of a programming language that perform different computations or actions depending on whether a programmer-specified Boolean condition evaluates to true or false. This statement is generally used to control the code flow based on specific program requirements. In Python, the easiest conditional "if" statement can be defined as (Lutz, 2013):

```
if <expression>:
   <statement>
    …
```

The <expression> is a condition evaluated in a Boolean context. If <expression> is true, then <statement> is executed. If <expression> is false, then <statement> is skipped over and the next line of code is executed.

Example:

```
def main():
   x = 10
   y = 20
   if x < y:
       print("x < y is true")

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

Result:

```
x < y is true
```

The following code checks whether `x > y`.

```
def main():
   x = 20
   y = 10
   if x < y:
       print("x < y is true")
   if x > y:
       print("x < y is false")
```

Result:

```
x < y is false
```

The Else and Elif Clauses

If the “if” statement is not executed, another required code may need to be executed instead. In these cases, the “else” clause is used in combination with the “if” statement as below (Matthes, 2019).

```
if <expression>:
   <statement>
   …
else:
   <statement>
   ….
```

Example:

```
def main():
   x = 20
   y = 10
   if x < y:
       print("x < y is true")
   else:
       print("x < y is false")
```

Result:

```
x < y is false
```

The elif clause can be used at the same time for multiple conditions.

```
if <expression>:
   <statement>
   …
```

```
elif <expression>:
   …
elif <expression>:
   ….
else:
   <statement>
   ….
```

Example:

```
def main():
   x = 20
   y = 20
   if x < y:
       print("x < y is true")
   elif x > y:
       print("x > y is true")
   elif x == y:
       print("x = y is true")
   else:
       print("there is no condition for x and y")
```

Result:

```
x = y is true
```

Python code can be repeated using loops. Inside the loop, the lines of code can be executed many times depending on specific conditions.

The “for loop” is used to perform iteration over any data structure.

Example:

```
def main():
   ebooks = ["python", "perl", "ruby"]
   for item in ebooks:
       print("The eBook is: {}".format(item))

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

Result:

```
The eBook is: python

The eBook is: perl
```

```
The eBook is: ruby
```

You can control the number of iterations by using the function **“**range(start, stop, step)”. This function returns a sequence of numbers starting from 0 (by default) with increments of 1 (by default), ending at the stop number minus one. Below, we print the x values only.

Example:

```
def main():
   for x in range(6, 11):
       print(x)
```

Result:

```
6

7

8

9

10
```

The “while” loop repeats as long as a specific Boolean condition is true.

Example:

```
def main():
   ebooks = ["python", "perl", "ruby"]
   i = 0
   while ebooks[i] != "ruby":
       print(ebooks[i])
       i += 1
```

Result:

```
python

perl
```

Using the “break” statement, we can stop the loop before it ends.

Example:

```
def main():
   ebooks = ["python", "perl", "ruby"]
   for item in ebooks:
       print(item)
       if item == "perl":
           break
```

Result:

```
python

perl
```

The “continue” statement is used to skip the current block and return back to the loop.

Example:

```
def main():
   ebooks = ["python", "perl", "ruby"]
   for item in ebooks:
       if item == "ruby":
           break
       else:
           print(item)
           continue
```

Result:

```
python

perl
```

As you can see, the “if” statement will break the loop when the item is equal to “ruby,” otherwise it will continue.

Python includes a “pass” statement to define a null operation — when it is executed, nothing happens in the program. This statement is used for syntactical purpose when no code needs to be executed.

Example:

```
def print_no_one(value):
  if value == 1:
      pass
else:
   print(value)
```

Result:

If the value is equal to 1, nothing will happen, but if the value is not equal 1, it will be printed.

1.4 Input / Output

Functions

The “input()” function allows the program to interact with the user. The “print()” function can be used to output the information entered by the user (Lutz, 2013).

Example:

```
def main():
   zip_code = input("Enter your zip code: ")
   print("The entered zip code is: {}".format(item))

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

Result:

Enter your zip code: 97124 (this was entered by the user)

The entered zip code is: 97124 (this was printed by the function print())

File IO

We open files in Python for reading and writing data. The access mode defined in the “open(mode, [buffering])” function selects the action related to the file.

These modes are:

- “r” — read mode, which is used when the file is only being read.
- “w” — write mode, which is used to edit and write new information to the file (any existing files with the same name will be erased when this mode is activated).
- “a” — appending mode, which is used to add new data to the end of the file. New information is automatically appended to the end.
- “r+” — special read and write mode, which is used to handle both actions when working with a file.

To write to the file, the “write()” function is used.

Example:

```
def main():
   file_test = open("test.txt", "w")
   file_test.write( "Python is a great language.\nSure!\n")
   file_test.close()

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

Result in the test.txt file:

```
Python is a great language.

Sure!
```

The folder in which the test.txt file is created is the same place that the program runs. As you can see, the created file object “file_test” needs to be closed to release its pertaining memory resources.

The access mode “w” will overwrite any existing content on the file. The access mode “a” will append (add) the content at the end of the file.

Example:

```
def main():
   file_test = open("test.txt", "a")
   file_test.write("Add this new line to the end of the file test.txt")
   file_test.close()
```

Result in the test.txt file:

```
Python is a great language.

Sure!

Add this new line to the end of the file test.txt
```

To read from a file, we use the “read()” function.

Example:

```
def main():
   file_test = open("test.txt", "r")
   print(file_test.read())

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

Result:

```
Python is a great language.

Sure!

Add this new line to the end of the file test.txt
```

To loop through the file line by line, we can use the “for” statement.

Example:

```
def main():
   file_test = open("test.txt", "r")
   for item in file_test:
       print(item)
```

Result:

```
Python is a great language.

Sure!

Add this new line to the end of the file test.txt
```

If you would like to return one line from the file, the method “readline()” can be used. The general syntax of this method is “file.readline(size)” where the “size” parameter is optional. It represents the number of bytes from the line that will return. The default number is —1, which indicates the whole line.

Example:

```
def main():
   file_test = open("test.txt", "r")
   lines = file_test.readline(6)
   print(lines)
```

Result:

```
Python
```

As you can see so far, working with files in Python requires knowledge of their location path. Python has a “pathlib” library to deal with files and paths. To use this library, the path or file name need to be passed to the “Path()” object using forward slashes.

Example:

```
def main():
      folder_path = Path(r"C:/folder_path/eclipse-workspace/iubh_fernstudium/
      src/1.4_input_output")
      file_name = "test.txt"
      file_path_name = folder_path / file_name
      file_test = open(file_path_name, "r")
      for item in file_test:
       print(item)
```

Result:

```
Python is a great language.

Sure!
```

The contents of a text file can be read without having to mess with opening and looping the file:

Example:

```
def main():
      folder_path = Path(r"C:/folder_path/eclipse-workspace/iubh_fernstudium/
      src/1.4_input_output")
      file_name = "test.txt"
      file_path_name = folder_path / file_name
      print(file_path_name.read_text())
```

Result:

```
Python is a great language.

Sure!
```

The “Path()” object can explicitly convert a Unix path into a Windows-formatted path.

Example:

```
def main():
      folder_path = Path(r"C:/Users/ebonat/eclipse-workspace/iubh_fernstudium/
      src/1.4_input_output")
      file_name = "test.txt"
      file_path_name = folder_path / file_name
      path_on_windows = PureWindowsPath(file_path_name)
      print(path_on_windows)
```

Result:

```
C:\Users\ebonat\eclipse-workspace\iubh_fernstudium\src\1.4_input_output
\test.txt
```

The optional "buffering" argument in the "open(mode, [buffering])" function specifies the file's desired buffer size: 0 means not buffered, 1 means line buffered, and any other positive value means that a buffer of approximately that size (in bytes) should be used. A negative buffering means that the system default should be used. If omitted, the system default will be used.

Logging Data

Logging is an excellent programming tool for debugging and auditing your program. By logging data in the necessary places, you can debug errors easily and use it to analyze program performance for futures updates. In general, the data is logged in text file(s) and/or in database table(s). To log data in Python, we use the built-in logging module with the "import logging" statement.

Logging levels
The log messages have five levels of event severity:

- DEBUG — This is used to log messages during programming debugging. It is very often applied in order to see returner values of variables and functions.
- INFO — This is used to log any information messages during program running.
- WARNING — This is used to log warning messages. It is very often applied in order to see preventing information that may have occurred during program running.
- ERROR — This is used to log error messages that occurred during program running. In general, these messages include specific error information, error line of code, date and time that the error occurred, function and module file names where the error occurred, etc.
- CRITICAL — This is used to log critical errors. It is very often applied in order to see generic program crashes.

Basic configuration
These levels are defined in the basic configuration method basicConfig(**kwargs). This method has the following parameters:

- level: This is the event severity level (DEBUG, INFO, WARNING, ERROR, and CRITICAL).
- filename: This is the file path and name.
- filemode: If filename is defined, the file is opened in this mode. The default is a, which means append.
- format: This is the message format.

The following example demonstrates how to create a simple warning message using this method.

Example:
```
import logging

def main():
   logging.basicConfig(filename="app.log", filemode="w", format="%(name)s
   - %(levelname)s - %(message)s")
   logging.warning("This message gets logged as warning.")

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

Result in the file app.log:

```
root - WARNING - This message gets logged as warning.
```

Logging variable data
Most of the time, the programmers want to include dynamic information from your application in the logs. This technique is often used during a large debugging process without any end-user interaction with the program. If a variable data needs to be logged, its name can be passed as a parameter to the selected logging method.

Example:
```
import logging
name = “Python”
logging.error(f'{name} raised an error')
```

Result:

```
root - ERROR - Python raised an error
```

1.5 Modules and Packages

Modules
Simply, a module is a file consisting of Python code that can define functions, classes, and variables.

Modules and packages are part of the Python Modular Programming style (Kapil, 2019). Modular programming is the process of dividing large programming tasks into smaller, more manageable sub-units or modules. Individual modules can be put together as building blocks to develop large enterprise business applications. This type of programming has the following benefits:

- Simplicity — A module typically focuses on one relatively small aspect of the program. This makes development, debugging, and testing very easy.
- Maintainability — It is easy to update a single module and/or add new features without touching the whole program. This will enable further collaboration between team members.
- Reusability — Functionality developed in a single module can easily be reused by other parts of the program. This eliminates the need to create duplicate code.
- Scoping — Modules are typically defined as separate namespaces, which helps to avoid conflicts between identifiers in different part of a program.

A module is a simple file with Python code and extension ".py". This file allows you to organize your code using variables, functions, and classes. We can save any code provided in this unit in a file and use it as a module.

Modules can be processed by using the following two statements and one important function:

- import — This allows a program (importer) to fetch a module as a whole entity.
- from — This allows programs to fetch particular names (functions, for example) from a module.
- imp.reload — This provides a way to reload a specific module's code without stopping Python.

In general, these statements are defined at the top of the calling modules, for example: "import os". This will make functions available that are located in the operating system (os) module (Beazley & Jones, 2013).

We will now look at the following math library module example. Create two files with following code:

File math.py

```
pi = 3.14159

def area_of_circle(radius):
   """
   calculate the area of a circle
   :param radius: radius of a circle
   :return: area of a circle
```

```
    """
    area = pi * (radius ** 2)
    return area
```

File client.py

```
import math_library

def main():
    result = math_library.area_of_circle(2)
    print("The area of the circle is: {}".format(result))

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

If you run the client.py file, the following result will be shown:

Result:

```
The area of the circle is: 12.56636
```

Importing the math library file to any other module will allow you to use the variables, functions, and classes defined within it. This is the simple idea behind using module libraries in Python. You can also execute the same idea using class objects.

Packages in Python are used to hierarchically organize module namespace. These packages are declared using the dot notation or from-import statement at the top of the module code. Both of these declarations will be shown in the sample code below. They are very popular in business application design and development today. If you move the math.py file to a package folder, math_packages, to run the client.py module, the following changes are required.

```
import math_packages.math_library

def main():
    result = math_packages.math_library.area_of_circle(2)
    print("The area of the circle is: {}".format(result))

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

Or:

```
from math_packages import math_library

def main():
```

```
    result = math_library.area_of_circle(2)
    print("The area of the circle is: {}".format(result))

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

Both programs produce the same result:

Result:

```
The area of the circle is: 12.56636
```

As you can see from the code above, the package can be imported as:

```
import <package_name>.<module_name>
from <package_name> import <module_name>
```

The first line imports the whole package. The second line only imports the selected <module_name>.

Summary

This unit covers the main data structures in Python, including primitive and non-primitive. The function definition and code are provided along with built-in and user-defined functions. This is built on as function default parameters, double-asterisk parameters, and single-asterisk parameters are explained. Additionally, variable scopes are covered, including a thorough explanation of local and global variables. To further build on basic Python knowledge, specific functions such as nested and global are presented and program flow control is provided with many different types of conditional implementations. Python files manipulation is presented with the latest used core libraries. Logging is an excellent programming tool for debugging and auditing your program. This topic is covered using Python best practices for data logging. Finally, Python code organization is explained, along with modules and packages design and implementation.

Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!

Unit 2

Classes and Inheritance

STUDY GOALS

On completion of this unit, you will have learned...

- ... how to define and use different types of variable scope.
- ... how to design and develop class objects.
- ... how to implement class inheritance and apply method overriding.
- ... how to create iterators and generators.

2. Classes and Inheritance

Introduction

The object-oriented programming (OOP) architecture is a very popular application design pattern today. It provides a way to structure programs so that properties and methods are bound to individual objects. OOP models real-world entities as software objects that have some data associated with them and can perform specific functions.

In this unit, we will cover the concepts of class and object where a class is a “blueprint” of a created object and an object is an instance of a class. The constructor method of a class will be presented with and without default values. Class inheritance implementation will be covered based on parent and child classes’ definition. To overwrite parent class methods, overriding implementation will be provided.

The concepts of Python scopes and namespaces will be presented including global, local, and built-in types. For loop objects will be covered including integrators and generators. The iteration tool module will be used to demonstrate how to speed-up sequence looping in Python.

2.1 Scopes and Namespaces

Name in Python

In Python, a name is a simple way to access any variable. A variable is declared by assigning a name to it. Below are some examples.

Name to reference values:

```
person_name = “John”
person_age = 63
person_weight = 87.5
person_is_old = True
```

Name to a function:

```
def my_function():
   print(“hello function”)

hello_function = my_function
```

Namespace
The fundamental idea of the **namespace** is to have a well-structured and organized code. Namespace is a simple dictionary structure where the keys represent the names and the values represent the object itself. It controls the relationship between names

and objects in the program, meaning that all names are unique in the program, they can exist independently from each other, and they are structured in a certain hierarchy (Lutz, 2013).

Namespace
A namespace is a system used to make sure that all the names in a program are unique and can be used without any conflict.

Here is an example of a namespace for two used names:

```
namespace_example = {"name_1":object_1, "name_2":object_2}
```

The following is a list of the used namespaces in Python:

1. Global namespaces — They are based on imported module definition(s) used in the program. The namespace is created for every module. It is available until the program ends.
2. Local namespaces — They are based on local names inside a function. The namespace is created or every function is called in a program. It is available until the function returns.
3. Built-in namespaces — They are based on built-in functions and built-in exception names. The namespace is created until the interpreter starts, it will then keep it until you exit. These are the outermost level of the Python language.

Scope

The parts of the program where you could use a name without any prefix is defined as scope (Ramalho, 2015). The scope of variables in Python are essentially references or pointers to an object in memory. When you assign a variable with = to an instance, you are binding (or mapping) the variable to that instance. Multiple variables can be bound to the same instance. In general, Python keeps track of all these mappings with namespaces. They are containers for mapping names of variables to objects. You can think of them as dictionaries, containing name:object mappings. This enables access to objects by using the names you choose to assign to them (Lutz, 2013).

There are four different types of scopes used for locals, functions, modules, and built-ins.

1. Local scope (innermost scope) — This defines all local names available in the current function.
2. Outermost scope — This defines the list of all built-in names in the whole program.
3. Module scope — This defines all global names from the current module.
4. Enclosing functions scope — This defines a name from the nearest enclosing scope and goes outwards.

We will now look at the following generic example:

```
def scope_testing():

   def local_scope():
       spam = "local spam defined"

   def nonlocal_scope():
```

```
        nonlocal spam
        spam = "nonlocal spam defined"

    def global_scope():
        global spam
        spam = "global spam defined"

    spam = "testing the spam"
    local_scope()
    print("After local scope test:", spam)
    nonlocal_scope()
    print("After nonlocal scope test:", spam)
    global_scope()
    print("After global scope test:", spam)

def main():
    scope_testing()
    print("In global scope:", spam)

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

Result:

```
After local scope test: testing the spam

After nonlocal scope test: nonlocal spam defined

After global scope test: nonlocal spam defined

In global scope: global spam defined
```

As you can see, the `local_scope()` function did not change the `scope_testing()` binding of spam variable. The `nonlocal_scope()` function changed the `scope_testing()` binding of spam variable. The `global_scope()` function changed the module-level binding.

2.2 Classes and Inheritance

Object-oriented programming (OOP) is a programming design pattern that provides a way of structuring programs so that properties and methods are bundled into individually designed objects (Matthes, 2019). OOP is all about increasing the ability of applications to reuse code and make it easier to write and understand. The encapsulation that

OOP provides allows developers to treat code as a black box. Using OOP features like inheritance makes it easier to expand the functionality of existing code and considerably reduce program development and testing time (Lutz, 2013).

Python is an OOP language; the program is built around objects that combine data and functionality. Everything in Python is an object with properties (attributes) and methods (functions). One of the most important concepts of OOP is the difference between classes and objects (Beazley & Jones, 2013).

- A class provides a building block of binding data and functionality together. A class is a "blueprint" of a created object.
- An object is an instance of a class. Objects have member variables and behavior is associated with them.

Class

A class is created by the **class** keyword.

Class
Creating a new class creates a new type of object, allowing new instances of that type to be made. Classes provide a way to bundle data and functionality together.

```
class ClassName:
   <statement>
   …
```

The following code is used to create a TextBook class.

```
class TextBook():
   '''
   docstring - uses to document the class
   '''
```

The `text_book` object is created as an instant of the TextBook class.

```
text_book = TextBook()
```

If you add a `print_book_title` method, the class code will be as follows:

```
class TextBook():

   def print_book_title(self, book_title):
       print(book_title)
```

As you can see, the first parameter of every class method is defined with the "self" keyword. It is always a reference to the current instance of the used class. In our case, "self" parameter guarantees that the method print_book_title() has a way of referring to object text_book attributes. Below is an example of how to use this class.

Example:

```
def main():
  #    object text_book is the instant of the class TextBook():
   text_book = TextBook()
   book_title = "Programming with Python"
  #    calling print_book_title() method
   text_book.print_book_title(book_title)

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

Result:

```
Programming with Python
```

The Constructor Method

A constructor is a special method of a class or structure in object-oriented programming that initializes an object of that type. A constructor is an instance method that usually has the same name as the class and can be used to set the values of the members of an object, either to default or to user-defined values. The constructor method of a class is initialized as soon as an object of a class is instantiated (Lutz, 2013).

This method is defined with the "`__init__`" reserve method in Python.

```
def __init__(self):
   '''
   class constructor
   '''
```

The following code creates a constructor and passes the number of pages of the book.

```
def __init__(self, pages_number):
     self.pages_number = pages_number
```

Example:

```
class TextBook():

   def __init__(self, pages_number):
     self.pages_number = pages_number

   def print_book_title(self, book_title):
       print(book_title)

   def print_book_pages_number(self):
```

```
        print("The number of pages is: {}".format(self.pages_number))

def main():
   pages_number = 300

  #    object text_book is the instant of the class TextBook():
   text_book = TextBook(pages_number)

   book_title = "Programming with Python"
   #    calling print_book_title() method
   text_book.print_book_title(book_title)

  #    calling print_book_pages_number method
   text_book.print_book_pages_number()

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

Result:

```
Programming with Python

The number of pages is: 300
```

If you do not pass the pages_number when the object text_book is instantiated, the following error will occur.

```
TypeError: __init__() missing 1 required positional argument: 'pages_number'
```

To avoid this error, you need to pass the number of pages as an optional parameter to the contractor and set it as equal to “None”.

```
def __init__(self, pages_number=None):
     if pages_number is not None:
         self.pages_number = pages_number
     else:
       self.pages_number = None
```

After this update, the final code will be as follows:

```
class TextBook():

   def __init__(self, pages_number=None):
     if pages_number is not None:
         self.pages_number = pages_number
     else:
```

```
        self.pages_number = None

    def print_book_title(self, book_title):
        print(book_title)

    def print_book_pages_number(self):
        if self.pages_number is not None:
            print("The number of pages is: {}".format(self.pages_number))

def main():
    pages_number = 300

    #    object text_book is the instant of the class TextBook():
    text_book = TextBook()

    book_title = "Programming with Python"
    #    calling print_book_title() method
    text_book.print_book_title(book_title)

   #    calling print_book_pages_number method
    text_book.print_book_pages_number()

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

Result:

```
Programming with Python
```

As you can see, the best way to pass optional parameters to a function in Python is by using the “None” keyword. Using this technique will reduce the amount of functions that must be written.

Class Inheritance

In computer programming, inheritance is a powerful feature in object-oriented programming (OOP). Inheritance allows computer programmers to create a class that inherits all the methods and properties from another class (Beazley & Jones, 2013). There are two elements in inheritance implementation.

1. Parent class — This is the class that is being inherited from, also called base or superclass class.
2. Child class — This is the class that inherits from another class, also called derived or subclass class.

One of the main benefits of inheritance is to minimize the amount of duplicate code where the general code is in the base class and multiple derived classes have access to it. This is a huge advantage for better code organization and future required updates. Below is a general class inheritance implementation code.

```
class BaseClassName(object):
   <statement>
   …
class DerivedClassName(BaseClassName):
   <statement>
   …
```

As you can see, in Python, any created class (BaseClassName) inherits from a generic class “object”. On the other hand, the derived class (DerivedClassName) inherits from the base class (BaseClassName). In the quality of class inheritance example, we will create three classes. Person (base class), Employee, and Customer (derived classes).

```
class PersonClass(object):

   def __init__(self, first_name, last_name):
       self.first_name = first_name
       self.last_name = last_name

   def print_name(self, person_type):
            print("The " + person_type + " name is: " + self.first_name + " "
            + self.last_name)

   def get_name_email(self, email):
       name_email = "The email of " + self.first_name + " "
            + self.last_name + " is: " + email
       return name_email

   def is_employee(self):
       return True

class EmployeeClass(PersonClass):

   def __init__(self, first_name, last_name):
       super().__init__(first_name, last_name)

   def print_employee_id(self, id):
       print("Employee " + self.first_name + " " + self.last_name + " id is: " + str(id))

   def is_employee(self):
       return True

class CustomerClass(PersonClass):
   def __init__(self, first_name, last_name):
```

```
        super().__init__(first_name, last_name)

    def print_customer_phone(self, customer_phone):
        print("Customer " + self.first_name + " " + self.last_name +
              " phone is: " + customer_phone)

    def is_employee(self):
        return False
```

The inheritance organization for these three classes is as follows:

- PersonClass(object) is inherited from class object.
- EmployeeClass(PersonClass) is inherited from PersonClass class.
- CustomerClass(PersonClass) is inherited from PersonClass class.

In the constructor of the EmployeeClass and CustomerClass classes, the super() function was used. It will enable these classes to inherit all the methods and properties from the PersonClass base class. This function returns a proxy object with the ability to call methods and use properties of the base class via delegation. Below is an example using these classes.

Example:

```
def main():
  #    print employee name
    first_name = "John"
    last_name = "Smith"
    employee_class = EmployeeClass(first_name, last_name)
    person_type = "employee"
    employee_class.print_name(person_type)

  #    print employee id
    employee_id = 100
    employee_class.print_employee_id(employee_id)

    #    check if an employee is employee
    result = employee_class.is_employee()
    print(result)

  #    print customer name
    first_name = "David"
    last_name = "Johnson"
    customer_class = CustomerClass(first_name, last_name)
    person_type = "customer"
    customer_class.print_name(person_type)

  #    print customer phone number
    customer_phone = "+1.800.503.987.6543"
    customer_class.print_customer_phone(customer_phone)
```

```
    #    check if a customer is employee
    result = customer_class.is_employee()
    print(result)

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

Results:

The employee name is: John Smith

The employee ID of John Smith is: 100

True

The customer name is: David Johnson

The phone number of customer David Johnson is: +1.800.503.987.6543

False

Method Overriding

A class can also create a completely new implementation in any base class method. In OOP terminology, this is called method overriding. Overriding is the property of a class to change the implementation of a method provided by one of its base classes. Overriding is a very important part of OOP since it makes inheritance utilize its full power. By using method overriding, a class may clone another class, avoiding duplicated code and, at the same time, enhancing or customizing part of it (Lutz, 2013).

In our previous inheritance code example, all three classes have an "`is_employee()`" method. This method returns "True" in the EmployeeClass class and "False" in the CustomerClass class. In the base PersonClass class, "True" is returned by definition. In this case, as you can see from the result of the program, the derived class CustomerClass overrides the "`is_employee()`" method in base class PersonClass. This method has no effect in any derived class.

2.3 Iterators and Generators

Iterable Objects

Any object in Python that can be used with a “for loop” or “while loop” statement is called an iterable object (Beazley & Jones, 2013).

Example:

```
def main():
   programming_language = ["JavaScript", "Python", "R", "Ruby",
"PHP", "Java", "C#", "C", "C++", "Julia", "Go", "Perl"]
   for item in programming_language:
       print(item)

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

Result:

```
JavaScript
Python
R
Ruby
PHP
Java
C#
C
C++
Julia
Go
Perl
```

This program prints every item in the programming language list object. Python has several built-in iterator objects such as lists, tuples, strings, dictionaries, and files. Iterators provide clear and well-organized code (Kapil, 2019).

Iter() function

Python supports a built-in function “iter”. This function takes an iterable object and returns an **iterator.**

Iterator
An iterator in Python is any Python type that can be used with a “for in loop.”

Example:

```
def main():
   number_list = [1, 2, 3, 4, 5]
   number_list_iter = iter(number_list)
  #    print 1
   print(next(number_list_iter))
  #    print 2
   print(next(number_list_iter))
   #    print 3
   print(next(number_list_iter))
   #    print 4
   print(next(number_list_iter))
  #    print 5
   print(next(number_list_iter))
  #    an error occurred - StopIteration
   print(next(number_list_iter))

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

Result:

```
1

2

3

4

5

main()

print(next(number_list_iter))

StopIteration
```

The “next of the iterable object” method returns the item (element). If there are no more elements, the exception error “StopIteration” will arise.

Iterators

Iterators are implemented as classes using the iterator protocol definition. An iterator protocol is a specific class that implements the methods `__iter__()` and `__next__()`.

- `__iter__` returns the iterator object itself.
- `__next__` returns the next iteration value.

Example:
Class file `iterator_class.py` code.

```
class SequenceNumber():

   def __init__(self, low, high):
       self.low = low
       self.high = high

   def __iter__(self):
       return self

   def __next__(self):
       if self.low < self.high:
           self.low += 1
           return self.low - 1
       else:
           raise StopIteration
```

File `iterator_call.py` code.

```
from iterator_class import SequenceNumber

def main():
   sequence_number = SequenceNumber(0, 3)
  #    print 0
   print(next(sequence_number))
  #    print 1
   print(next(sequence_number))
  #    print 2
   print(next(sequence_number))
  #    an error occurred: StopIteration
   print(next(sequence_number))

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

Result:

```
0

1

2

Traceback (most recent call last):

File "class_path\iterator_class.py", line 23, in __next__

raise StopIteration

StopIteration
```

The result above shows the iteration error “StopIteration” when the iterator object “sequence_number” was trying to move to the next item.

Generators

A generator is an object that allows you to create iterators in an easy way. You do not need to use a customized class with methods __iter__() and __next__() to create a generator. A simple function without a return statement and the “yield” keyword can create a generator.

Example:

```
def sequence_number_generator(low, high):
   while low <= high:
      yield low
      low += 1

def main():
   number_list = []
   for number in sequence_number_generator(0, 5):
       number_list.append(number)
   print(number_list)

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

Result:

```
[0, 1, 2, 3, 4, 5]
```

As we can see, the function sequence_number_generator() contains a yield statement inside the while loop. When the loop starts and reaches the yield statement, the value of the local variable low is returned and the generator stops. When the next call occurs, the generator starts again from the value at which it previously stopped and increases this value by one. In general, the yield statement provides the result of the function simulating the return statement and keeps the value of the local variable low.

Itertools Module

Python has an "itertools" module that provides built-in efficient iterator functions for high-speed looping (Beazley & Jones, 2013). For example, we will look at the count(), cycle(), and repeat() functions.

1. count(start=0, step=1) — Make an iterator that returns evenly spaced values starting with number start.
   Example:

```
from itertools import count

def main():
   sequence = count(start=2.5, step=0.5)
   while(next(sequence) <= 8):
       print(next(sequence))

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

Result:

```
3.0
4.0
5.0
6.0
7.0
8.0
```

2. cycle(iterable) — Make an iterator returning elements from the iterable and saving a copy of each.
   Example:

```
from itertools import cycle

def main():
   person = cycle(["employee","customer"])
   count = 0
   while(count != 6):
       print(next(person))
```

```
        count+=1

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

Result:

```
employee
customer
employee
customer
employee
customer
```

3. repeat(object[, times]) — Make an iterator that returns object over and over again. It runs indefinitely unless the times argument is specified.
   Example:

```
from itertools import repeat

def main():
   person = repeat(object=1000, times=3)
   count = 0
   while(count != 3):
       print(next(person))
       count+=1

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

Result:

```
1000
1000
1000
```

Summary

The fundamental idea of the Python namespace is to have a well-structured and organized code. This unit covers the main namespaces in Python, including global, local, and built-in. It defines the scope as the parts of the program where you could use a name without any prefix.

The four different types of scopes used for locals, functions, modules, and built-ins are presented.

The definition and benefits of object-oriented programming (OOP) as a programming design pattern are covered, for example, it provides a means of structuring programs so that properties and methods are bundled into individual designed objects. The concept of class as a building block of binding data and functionality together is also explained and the object as an instance of a class is defined. The definition of the class constructor and its implementation is provided with generic programming code.

Class inheritance is explained, including the fundamental definition of classes as Parent and Child. The derived class can also create a completely new implementation to any base class method. This is call method overriding, which is presented in detail in this unit. Also covered are iterators and generators, which are objects that can be used with “for loop” or “while loop” statements. The itertools module, which provides built-in efficient iterator functions for high-speed looping, is also shown in this unit.

Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!

Unit 3

Errors and Exceptions

STUDY GOALS

On completion of this unit, you will have learned...

... how to define program errors and get the exception information.

... how to handle and raise exceptions.

... how to create custom user-defined exceptions classes.

3. Errors and Exceptions

Introduction

Writing programs that work when everything goes as expected is a fundamental principle of software development. Sometimes errors occur and the program needs to be smart enough to handle them. The process of responding to these errors during program execution is called exception handling.

Error handling is generally resolved by saving the state of execution at the moment the error occurred and interrupting the normal flow of the program to execute a special function or piece of code, which is known as the exception handler.

Exception handling is an important implementation for any computer program. In general, exception handling in software is error checking, which maintains normal program flow while carrying out explicit checks for contingencies which are then reported using parameters of the special return exception information.

Exception handling is implemented by using built-in programming language exception information objects and/or by raising (creating) your own custom exceptions. Both implementations are very useful and they can be applied to any possible required error checking in your program. In general, built-in exceptions information is used to catch generic program execution errors, whereas raise exceptions are implemented to aid data entry and program conditional validations.

Python programming language handles exception errors using the system module with the general exception information function. Python has a well-defined code block organization that includes try, except, and finally. The except block takes care of all occurred program execution exception errors. The custom development of user-defined functions allows software engineers to create their own custom exception libraries.

3.1 Syntax Errors

Syntax errors
The most common reason that an error occurs in a Python program is because a certain statement is not in accordance with the prescribed usage.

In computer programming, three types of errors may occur: **syntax errors**, runtime errors, or logic errors. The syntax errors occur when a program does not conform to a specific statement of a programming language, and the compiler or the interpreter cannot run the source code. The runtime errors occur when a program starts running or during program execution. The logic errors occur when a program does not follow the predefined specific requirements that must be implemented.

Python Error Examples

We will now look at some Python error examples.

Example of a syntax error:

```
def main():
   print(1/1))

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

Result:

```
print(1/1))
            ^
SyntaxError: invalid syntax
```

Example of a runtime error:

```
def main():
   print(1/0)

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

Result:

```
print(1/0)

ZeroDivisionError: division by zero
```

Example of a logic error:

```
def main():
   a = "Python"
   if a == "Python":
       print("a is equal Java")

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

Result:

```
a is equal Java
```

This result should be:

```
a is equal Python
```

Exception Information

An exception is an error that happens during the execution of a program. When that error occurs, Python generates an exception that can be handled, preventing a crash. Exceptions are convenient in many ways for handling errors and special conditions in a program. Exception handling is a must in any computer program (Lutz, 2013).

Now we will find out how to get the exception information in Python. The built-in `sys.exc_info()` function returns a tuple of three values (type, value, traceback) that all give information about the exception that is currently being handled. These parameters mean the following:

- type — This finds out the type of the exception being handled.
- value — This finds out the exception instance type.
- traceback — This finds a traceback object that encapsulates the call stack at the point where the exception originally occurred.

Here is the line of code needed to get these values.

```
exception_type, exception_value, exception_traceback = sys.exc_info()
```

Traceback
As explained before, the traceback object contains the information of the statement calls made in your code at a specific line. This object returns a tuple of the following four exception parameters: file name, line number, procedure name, and line code.

```
file_name, line_number, procedure_name, line_code = traceback.extract_tb
(exception_traceback)[-1]
```

We will look at the division by zero runtime error.

Example:

```
import sys
import traceback

def main():
   division_by_zero = 1/0
   print(division_by_zero)

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

Result:

```
division_by_zero = 1/0
```

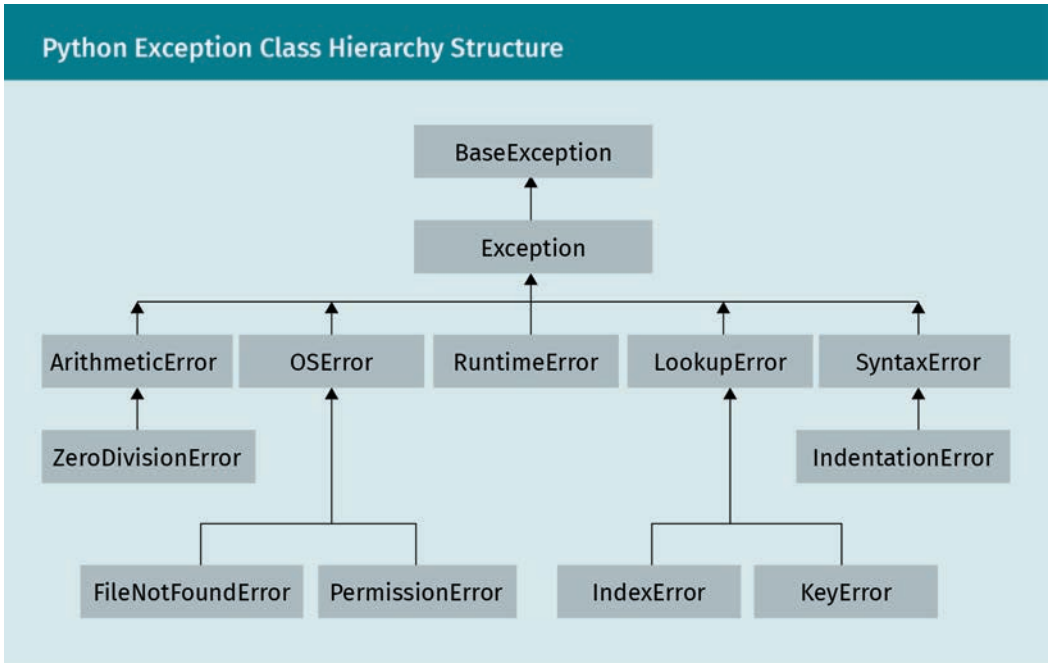
```
ZeroDivisionError: division by zero
```

In this case, the exception type is “ZeroDivisionError” and the exception value is “division by zero.” Python does not show the traceback information by default when an exception occurs.

Creating custom exceptions

Python allows software engineers to create custom exception classes. This implementation can be done by following the exception class hierarchy structure shown in the following graphic. As you can see, the “BaseException” class is the root of all exception classes in Python. The exception is never raised on its own and should instead be inherited by other, lesser exception classes that can be raised. The “Exception” class is the most commonly inherited exception type for creating a custom exception class. In general, all exception classes that are considered errors are subclasses of the Exception class (Beazley & Jones, 2013).

One important child subclass of the Exception class is the “LookupError” class. This class is the base class for “IndexError” and “KeyError” classes, which define, of course, the index and key errors information. We can finalize that custom exception classes should be derived from Exception and IndexError built-in classes. The Exception class is a built-in class and non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class. The IndexError class is a built-in class. Exceptions from this class are raised when a sequence subscript is out of range.



3.2 Handling and Raising Exceptions

In Python, like many programing languages, the exceptions are handled using the try-except-finally blocks. Below is an example of this general block structure (Matthes, 2019).

```
def handle_exceptions():
   try:
       pass
      # function code
   except:
       pass
      # catch and handle exceptions
   finally:
       pass
      # code clean-up. this block always occurs

def main():
   handled_exceptions()

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

Try block
The try and except block in Python is used to catch and handle exceptions.

Python executes the code in the **try block**. The code that follows the except statement is the program's response to any exceptions in the preceding try clause. As demonstrated earlier, when syntactically correct code runs into an error, Python will throw an exception error. If an exception occurs, the except block is triggered. The finally block always occurs regardless of whether or not an exception occurs.

Now, we will apply this code to our division by zero runtime error case.

```
def division_by_zero():
   try:
       division_by_zero = 1/0
   except:
       exception_type, exception_value, exception_traceback = sys.exc_info()
       print("Exception Type: {}\nException Value: {}".format(exception_type,
            exception_value))
       file_name, line_number, procedure_name, line_code = traceback.extract_tb
(exception_traceback)[-1]
       print("File Name: {}\nLine Number: {}\nProcedure Name: {}\nLine Code:
            {}".format(file_name, line_number, procedure_name, line_code))
   finally:
       pass

def main():
   division_by_zero()
```

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

Result:

```
Exception Type: <class 'ZeroDivisionError'>
Exception Value: division by zero
File Name: C:\file_path\iubh_fernstudium\src\3.1_syntax_errors\errors_exceptions.py
Line Number: 26
Procedure Name: division_by_zero
Line Code: division_by_zero = 1/0
```

As you can see, all of the exception information has been defined including exception class type and value, file name, line number, procedure name, and line of code where the exception occurred. In a real business application development, it makes sense to create a generic exception handling function that can be reused in all custom developed functions. The function get_exception_info() can provide this implementation.

```
def get_exception_info():
   try:
       exception_type, exception_value, exception_traceback = sys.exc_info()
       file_name, line_number, procedure_name, line_code = traceback.extract_tb
(exception_traceback)[-1]
exception_info = ''.join('[Time Stamp]: '
 + str(time.strftime('%d-%m-%Y %I:%M:%S %p'))
 + '' + '[File Name]: ' + str(file_name) + ' '
 + '[Procedure Name]: ' + str(procedure_name) + ' '
 + '[Error Message]: ' + str(exception_value) + ' '
 + '[Error Type]: ' + str(exception_type) + ' '
 + '[Line Number]: ' + str(line_number) + ' '
 + '[Line Code]: ' + str(line_code))
   except:
       pass
   return exception_info
```

As you can see, the time stamp has been included. In many cases it is important to know exactly when the exception occurred. If we apply this function in our case, then the final program code will be as follows:

```
import sys
import traceback
import time

def get_exception_info():
   try:
       exception_type, exception_value, exception_traceback = sys.exc_info()
       file_name, line_number, procedure_name, line_code = traceback.extract_tb
```

```
(exception_traceback)[-1]
exception_info = ''.join('[Time Stamp]: '
 + str(time.strftime('%d-%m-%Y %I:%M:%S %p'))
 + '' + '[File Name]: ' + str(file_name) + ' '
 + '[Procedure Name]: ' + str(procedure_name) + ' '
 + '[Error Message]: ' + str(exception_value) + ' '
 + '[Error Type]: ' + str(exception_type) + ' '
 + '[Line Number]: ' + str(line_number) + ' '
 + '[Line Code]: ' + str(line_code))
   except:
       pass
   return exception_info

def division_by_zero():
   try:
       division = 1/0
   except:
       exception_info = get_exception_info()
       print(exception_info)
   finally:
       pass

def main():
   division_by_zero()

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

Result:

```
[Time Stamp]: 20-09-2019 08:37:07 AM [File Name]: C:\Users\ernest\eclipse
workspace\iubh_fernstudium\src\3.1_syntax_errors\errors_exceptions.py
[Procedure Name]: division_by_zero [Error Message]: division by zero
[Error Type]: <class 'ZeroDivisionError'> [Line Number]: 26
[Line Code]: division = 1/0
```

Using this complete exception information will allow application developers to quickly find and fix their program bugs.

The Else Clause

The else clause, in handling exception block logic, enables the execution of a certain block of code if the exception does not occur.

```
def handle_exceptions():
   try:
       pass
```

```
        # function code
    except:
        pass
       # catch and handle exceptions
    else:
        pass
        # no exceptions occur, run this code
    finally:
        pass
       # code clean-up. this block always executes

def main():
    handled_exceptions()

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

Now, we will apply this else clause to our case.

```
def division_by_zero():
    try:
        division = 1/1
    except:
        exception_info = get_exception_info()
        print(exception_info)
    else:
        print("Nice! no exceptions occured. Write some code here.")
    finally:
        pass

def main():
    division_by_zero()

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

Result:

```
Nice! no exceptions occured. Write some code here.
```

Raising an Exception

The **raise statement** does two things: It creates an exception object, and immediately leaves the expected program execution sequence to search the enclosed try statements for a matching except clause. The raise statement allows us to throw a specified

Raise statement
The effect of a raise statement is to either divert execution in a matching except suite, or to stop the program because no matching except suite was found to handle the exception.

exception to occur (Kapil 2019). This allows software engineers to create their own custom exception messages for specific program conditions or implementation requirements. In general, this statement can be used inside and/or outside the except block. In many user cases, this statement has been implemented for data control and validation. We will now look at our example logic error.

```
def main():
   a = "Java"
   if a != "Python":
       raise Exception("a is not equal Python")

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

Result:

```
raise Exception("a is not equal Python")

Exception: a is not equal Python
```

In the example above, a specific custom exception message can be defined and passed to the raise exception statement.

3.3 User-Defined Exceptions

As we know, exceptions are built-in class objects in Python. Based on that, we can inherit from them to create our own custom exception objects (Ramalho, 2015). Below are the generic MyException(Exception) and MyIndexError(IndexError) inherited classes from Exception and IndexError main base classes.

```
class MyException(Exception):
   def __init__(self,*args,**kwargs):
       super().__init__(self,*args,**kwargs)

class MyIndexError(IndexError):
def __init__(self,*args,**kwargs):
       super().__init__(self,*args,**kwargs)
```

We will now look at a simple example to understand how to create user-defined exception class objects. Suppose you would like to create a custom defined exception that catches a specific parameter and a message. In this case, your custom derived exception class will be as follows:

```
class MyException(Exception):
   def __init___(self, exception_parameter, exception_message):
       super().__init__(self, exception_parameter, exception_message)
```

The main() function that calls this user-defined exception will be

```
def main():
   programming_language = ["JavaScript", "Python", "R", "Ruby", "PHP", "Java",
 "C#", "C", "C++", "Julia", "Go", "Perl"]
   for item in programming_language:
       if item != "Python":
           raise MyException(item, "My exception was raised with exception argument:
 {}".format(item))

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

Result:

```
raise MyException(item, "My exception was raised with exception argument: {}".format(item))
__main__.MyException: ('JavaScript', 'My exception was raised with exception argument:
JavaScript')
```

Creating user-defined exception classes requires careful design and lots of effort. They are mostly used for the development of custom libraries.

Summary

Program syntax, runtime, and logic errors are defined in this unit, accompanied by source code to validate their definition. An exception is an error that happens during the execution of a program. Python programming language handles exception errors using the system module with the general exception information function. This function returns the exception type, exception value, and exception traceback object that contains specific information parameters.

This unit covers how to write the Python code to catch these exception errors. In Python, like other programming languages, the exceptions are handled using the try-except-finally blocks. The program block explanation and the source code are provided. The code to get any exception error that has occurred is provided in a generic function definition. The raise statement that allows us to throw a specified exception to occur is covered. This statement allows us to use any custom exception message. The Python exception class hierarchy structure was shown and explained. The creation of custom user-defined exception class objects was covered using Exception and IndexError class objects.

Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!

Unit 4

Python Important Libraries

STUDY GOALS

On completion of this unit, you will have learned...

- ... how to use the most popular Python Data Ecosystem libraries.
- ... how to perform scientific calculations using NumPy, pandas, and SciPy libraries.
- ... how to use the dynamic compiler Numba to speed up Python programs.
- ... how to visualize different types of data using Matplotlib, Seaborn, and Bokeh libraries.
- ... how to access MySQL database with SQLAlchemy and PyMySQL client driver.

4. Python Important Libraries

Introduction

The Python libraries include the default core libraries from the distribution package as well as the data ecosystem libraries that are mainly developed for data analytics and machine learning/artificial intelligence projects. These 1000+ developed libraries make Python one of the most popular programming languages today and, since all of the libraries are open sourced, we have added commits, contributors count, and other metrics from Github, which could serve as a proxy metric for library popularity. The most important and useful libraries for data scientists and data engineers, based on recent experience, include NumPy, pandas, matplotlib, SymPy, IPython, SciPy, Seaborn, scikit-learn, and Bokkeh. These libraries are constantly used in machine learning/artificial intelligence projects.

Speeding up Python programs is a very important task for software engineers and data scientists today. The most common libraries that can be used to speed up Python code are Cython, PyPy, and Numba. It is important to mention that Numba is currently the most popular program used to speed up Python programs. For data visualization, Python uses Matplotlib, Seaborn, and Bokeh libraries. Data visualization is an important step in any data statistical analysis or machine learning project. Accessing database engines using Python is a very important requirement of the design and development of database business applications today. The SQLAlchemy SQL toolkit is the most popular database library for these cases.

4.1 Standard Python Library

The Python standard library consists of reusable built-in module files that are imported in your programs. They are included in the core Python distribution package. Python supports 200+ standard libraries. These libraries are written in C language. Apart from this standard library, there is a huge international community of people and companies developing custom libraries for Python constantly. This mass of 1000+ developed libraries is part of the Python Data Ecosystem (Ramalho, 2015).

Here are the most popular data ecosystem libraries used in Python today:

- NumPy is known as Numerical Python. It is a fundamental package for scientific computing. This may be the most important library in Python programming language today.
- Pandas is an open source library providing high-performance, easy-to-use data structures, and data analysis tools.
- Matplotlib is a Python 2D plotting library that produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

- SymPy is a developed library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python.
- IPython is a rich architecture for interactive computing with a powerful interactive shell, a kernel for Jupyter, and support for interactive data visualization and use of GUI toolkits. It also includes flexible, embeddable interpreters to load into your own projects, and easy to use, high performance tools for parallel computing.
- SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering. Some of the core packages include NumPy, SciPy library, matplotlib, IPython, SymPy, and pandas.
- Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.
- Scikit-learn is the main machine learning framework. It is a simple and efficient tool for data mining and data analysis. This framework is built on NumPy, SciPy, and matplotlib libraries.
- Bokkeh is an interactive visualization library that targets modern web browsers. Its goal is to provide elegant, concise construction of versatile graphics and to extend this capability with high-performance interactivity over very large or streaming datasets.

4.2 Scientific Calculations

Scientific calculations are always necessary in research projects and business application development today (Beazley & Jones, 2013). Python is an excellent option for scientific calculations as it has a huge international community of users, it is easy to code and to find, and there is an extensive ecosystem of scientific and engineering. Python also has a reputation for good performance due to highly optimized codes written in C language with excellent software, hardware support for parallel processing, and no license cost as Python is an open source software associated with the Python Software Foundation License (PSFL) (Lutz, 2013).

NumPy Library

The NumPy library is used in almost all numerical computation in Python. We can say that NumPy could be the most important library for the whole Python Data Ecosystem package. It provides high-performance vector, matrix, and higher-dimensional data structures for Python. It writes in C and FORTRAN, meaning that when the math calculations are formulated with vectors and matrices, the performance is very high.

NumPy array
NumPy array is a central data structure of the Installing Python libraries. It provides a collection of tools for high-performance multidimensional array objects and is a powerful data structure for the efficient computation of arrays and matrices. To work with

these arrays, there is a vast amount of high-level mathematical functions that operate on the matrices and arrays. We will now look at the following one-dimensional array creation.

```
np_array = a = np.array([1, 2, 3, 4, 5])
print("create array from a list:")
print(np_array)
```

If we run this code the result will be as follows:

Result:

```
create array from a list:

[1 2 3 4 5]
```

On a structural level, an array is basically nothing but pointers. It is a combination of a memory address, a data type, a shape, and strides. These aspects of an array have the following qualities:

- The data pointer indicates the memory address of the first byte in the array.
- The data type or dtype pointer describes the kind of elements that are contained within the array.
- The shape indicates the shape of the array.
- The strides are the number of bytes that should be skipped in memory to go to the next element. If your strides are (100,1), you need to proceed one byte to get to the next column and 100 bytes to locate the next row.

In other words, an array contains information about the raw data, how to locate an element, and how to interpret an element.

There are two ways of installing Python libraries:

- PyPI (pip) package installer — To install, for example, a “library_name”, either “pip install library_name” or “python -m pip install --upgrade library_name” is required.
- “conda” command from Anaconda Distribution package — At the command prompt use “conda install library_name”.

To install NumPy, we can use “pip install numpy” or “conda install numpy”. If you installed Python using the Anaconda Distribution package, it is recommended to use the “conda” command. To import NumPy, the line “import numpy as np” is required.

Because this library is very big, some examples are provided below so that you can understand how to use it.

Examples:

```
import numpy as np

def main():
   print("numpy version: {}".format(np.__version__))

   np_array = a = np.array([1, 2, 3, 4, 5])
   print("create array from a list:")
   print(np_array)

   np_array = a = np.array((6, 7, 8, 9, 10))
   print("create array from a tuple:")
   print(np_array)

   np_shape = np_array.shape
   print("one-dimension array shape, 5 elements:")
   print(np_shape)

   np_array = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
   print("two-dimensions array:")
   print(np_array)

   np_shape = np_array.shape
   print("two-dimensions array shape, 2 rows and 5 columns:")
   print(np_shape)

   np_array = a = np.array((1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
   np_reshape = np_array.reshape(5, 2)
   print("returns the array with a modified shape, 5 rows and 2 columns:")
   print(np_reshape)

   np_dim = np_array.ndim
   print("array dimensions:")
   print(np_dim)

   np_size = np_array.size
   print("array size: column number x row number:")
   print(np_size)

   np_dtype = np_array.dtype
   print("data type of the elements in the array:")
   print(np_dtype)

   np_itemsize = np_array.itemsize
   print("the size in bytes of each element of the array:")
   print(np_itemsize)

   np_data = np_array.data
```

```
   print("the buffer containing the actual elements of the array.
don't need it for calculations")
   print(np_data)

   array_1 = np.array([1, 2, 3, 4, 5])
   array_2 = np.array([6, 7, 8, 9, 10])

   array_result = array_1 + array_2
   print("addition operation:")
   print(array_result)

   array_result = array_1 - array_2
   print("subtraction operation:")
   print(array_result)

   array_result = array_2 - array_1
   print("subtraction operation:")
   print(array_result)

   array_result = array_1 * array_2
   print("multiplication operation:")
   print(array_result)

   array_result = array_1 / array_2
   print("division operation:")
   print(array_result)

   np_corrcoef = np.corrcoef(array_1, array_2)
   print("pearson product-moment correlation coefficients")
   print(np_corrcoef)

   np_mean = np.mean(array_1)
   print("arithmetic mean:")
   print(np_mean)

   np_average = np.average(array_1)
   print("weighted average:")
   print(np_average)

   np_std = np.std(array_1)
   print("standard deviation:")
   print(np_std)

   np_median = np.median(array_1)
   print("median:")
   print(np_median)

   np_variance = np.var(array_1)
```

```
    print("variance:")
    print(np_variance)

    np_min = np.min(array_1)
    print("minimum:")
    print(np_min)

    np_max = np.max(array_1)
    print("maximum:")
    print(np_max)

    np_summa = np.sum(array_1)
    print("summa:")
    print(np_summa)

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

Results:

```
numpy version: 1.15.3
create array from a list:
[1 2 3 4 5]
create array from a tuple:
[ 6 7 8 9 10]
one-dimension array shape, 5 elements:
(5,)
two-dimensions array:
[[ 1 2 3 4 5]
 [ 6 7 8 9 10]]
two-dimensions array shape, 2 rows and 5 columns:
(2, 5)
returns the array with a modified shape, 5 rows and 2 columns:
[[ 1 2]
 [ 3 4]
 [ 5 6]
 [ 7 8]
 [ 9 10]]
array dimensions:
1
array size: column number x row number:
10
data type of the elements in the array:
int32
the size in bytes of each element of the array:
4
the buffer containing the actual elements of the array. don't need it for calculations
<memory at 0x000000000366B948>
addition operation:
```

```
[ 7 9 11 13 15]
subtraction operation:
[-5 -5 -5 -5 -5]
subtraction operation:
[5 5 5 5 5]
multiplication operation:
[ 6 14 24 36 50]
division operation:
[0.16666667 0.28571429 0.375     0.44444444 0.5]
pearson product-moment correlation coefficients
[[1. 1.]
 [1. 1.]]
arithmetic mean:
3.0
weighted average:
3.0
standard deviation:
1.4142135623730951
median:
3.0
variance:
2.0
minimum:
1
maximum:
5
summa:
15
```

Pandas Library

The pandas library is the main data structure and processing tool in Python. It is built on top of the NumPy library, meaning that a lot of NumPy structures are used or replicated in the pandas library. This can be used for data analysis and visualization. Pandas library has many data supports, including heterogeneous data types, easy and fast handling of missing data, generic code that is simple to write, labeled data, and relational data. This library is used extensively in machine learning projects for data load, preprocessing (cleansing), and profiling.

Pandas can be installed using “pip install pandas” or “conda install pandas” with the Anaconda Distribution package. In general, the pandas library is imported as “import pandas as pd” in a Python module file. Pandas library contains the following two main components:

1. Series — This is a one-dimensional labeled array that can hold any data type, for example, integers, strings, floating point numbers, Python objects, etc. The axis labels are collectively referred to as the index. A series can be created as

```
s_data = pd.Series(data, …)
```

whereas “data” can be Python dictionary, ndarray (n-dimensional array or any scalar value), etc.

Example:

```
import numpy as np
import pandas as pd

def main():
   print("pandas version: {}".format(pd.__version__))

  # randn(): return a sample (or samples) from the standard normal distribution
   s_data = pd.Series(np.random.randn(5))
   print(s_data)

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

Results:

```
pandas version: 0.25.0
0   1.520672
1   0.922902
2  -0.949219
3   0.405605
4   1.875153
dtype: float64
```

2. DataFrames — This is a two||-dimensional labeled data structure with rows and columns that have the same or different data types. It is similar to a Microsoft Excel spreadsheet or SQL database table objects.
The first column of this component is reserved for the index values. If these values are not defined, pandas will automatically generate integers starting with zero. This component is the most useful in machine learning projects for data manipulation today. Below is the code that demonstrates how to create a pandas DataFrame object:

```
df_data = pd.DataFrame(data, …)
```

The DataFrame can accept many different kinds of input data such as 1D or 2D NumPy arrays, lists, dictionaries, tuples, structured or record ndarray, a series, or another DataFrame.

Example:

```
import numpy as np
import pandas as pd

def main():
```

```
  # create a dataframe from a dictionary
   dictionary = {"col 1":[1., 2., 3., 4., 5.], "col 2":[6, 7, 8, 9, 10],
"col 3":["uno", "dos", "tres", "cuatro", "cinco"]}
   df_data = pd.DataFrame(data=dictionary)
   print(df_data)

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

Results:

DataFrame Results

	col1	col2	col3
0	1.0	6	uno
1	2.0	7	dos
2	3.0	8	tres
3	4.0	9	cuatro
4	5.0	10	cinco

The indexes can be defined with text row numbers.

```
   #     create a dataframe from a dictionary with text row numbers indexes
   df_data = pd.DataFrame(data=dictionary, index=["row1", "row2", "row3",
 "row4", "row5"])
   print(df_data)
```

The final results will be as follows:

Results:

DataFrame Final Results

	col1	col2	col3
row1	1.0	6	uno

DataFrame Final Results			
row2	2.0	7	dos
row3	3.0	8	tres
row4	4.0	9	cuatro
row5	5.0	10	cinco

It is very important to mention that axis labeling information in pandas objects serves many purposes such as data identification using defined indicators which are used for profiling, visualization, and analysis, enabling automatic and explicit data alignment, and fast data retrieval by getting and setting the subsets of the dataset.

One of the common tasks in scientific calculations for data analytics is to load data from a comma-separated value (CSV) file. This file delimits plain text data that uses a comma to separate values. Suppose you have a CSV file "`input_file_with_nan.csv`" and the data table shown below.

Data Table			
col1	col2	col3	col4
	2.1	bar	TRUE
			FALSE
3	2.3	club	TRUE
5	2.4	bar	
	2.5	bar	FALSE
6		club	TRUE
2	2.5	house	
3	2.3	club	TRUE

The pandas library can load this data into the DataFrame using the “read_csv()” method.

```
import pandas as pd
def main():
   df_input_file = pd.read_csv(filepath_or_buffer="input_file_with_nan.csv")
   print(df_input_file)

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

The result of this program is as follows:

Result of the Program				
	col1	col2	col3	col4
0	NaN	2.1	bar	TRUE
1	NaN	NaN	NaN	FALSE
2	3	2.3	club	TRUE
3	5	2.4	bar	NaN
4	NaN	2.5	bar	FALSE
5	6	NaN	club	TRUE
6	2	2.5	house	NaN
7	3	2.3	club	TRUE

It is important to note that pandas handles empty values with a float defined value as not-a-number (NaN). This value is not an empty string, a null value, nor equal to 0. Pandas library includes many methods that can be used to handle these values properly during data manipulation process. These values will not be included in any computing calculations.

To get a better idea of how the pandas library works for data preprocessing, we will now look at the following practical example. Suppose you have a CSV file “`input_data.csv`” as shown below.

CSV File							
one	two	three	four	five	six	seven	eight
	2.1	3.1	bar	TRUE		£200.45	BMC
		3.2		FALSE	1/1/2017	£650.38	ACS
3	2.3	3.3	club	TRUE	2/1/2017	£85.07	ACS
5	2.4		bar		3/1/2017		BPS
	2.5	3.5	bar	FALSE		£230.22	BMC
6		3.3	club	TRUE	2/1/2017	£650.96	CJS
2	2.5		house		4/1/2017		FM
3	2.3	3.3	club	TRUE	2/1/2017	£85.07	ACS

Based on this file, the data preprocessing requirements document will include the following points:

- Drop duplicated rows and keep the first.
- Replace empty values with the mean value in columns 1 and 2.
- Replace empty values with the median value in column 3.
- Replace empty values with the “club” string in column 4.
- Replace empty values with the true boolean in column 5.
- Replace empty values with the current date in column 6.
- Replace empty values with the £0.0 in column 7.
- Remove the first character (English pound symbol) in column 8.
- Substitute the abbreviation in column 8.
- Define x column labels.
- Define y column target.
- Save the final file as “`output_data.csv`”. File location path must be defined.

Below is the code implementation for this document using pandas and NumPy libraries.

```
# -*- coding:latin1 -*-

import os
import sys
import datetime
```

```
import pandas as pd
import numpy as np
import config

def main():
   print("using pandas library for data manipulation and cleansing:")
   print()

   project_directory_path = os.path.dirname(sys.argv[0])
   print("project directory path:")
   print(project_directory_path)
   print()

   input_file_path = os.path.join(project_directory_path,
   config.INPUT_FILE_PATH)
   print("input file path:")
   print(input_file_path)
   print()

   output_file_path = os.path.join(project_directory_path,
   config.OUTPUT_FILE_PATH)
   print("output file path:")
   print(output_file_path)
   print()

   df_input_file = pd.read_csv(filepath_or_buffer=input_file_path, sep=",",
   encoding="latin1")
   print("pandas data frame for input file:")
   print(df_input_file)
   print()

   print("number of rows and columns:")
   print(df_input_file.shape)
   print()

   print("file information:")
   df_input_file.info()
   print()

   print("summary descriptive stats for numerical columns:")
   print(df_input_file.describe())
   print()

   df_result = df_input_file["four"].value_counts()
   print("frequency distribution for categorical column four:")
   print(df_result)
   print()
```

```
df_result = df_input_file["eight"].value_counts()
print("frequency distribution for categorical column eight:")
print(df_result)
print()

df_result = df_input_file.apply(lambda x: sum(x.isnull()), axis=0)
print("missing values (nan) by columns:")
print(df_result)
print()

df_result = df_input_file.duplicated()
print("duplicated rows:")
print(df_result)
print()

df_input_file = df_input_file.drop_duplicates(keep="first")
print("drop duplicated rows and keep the first:")
print(df_input_file)
print()

df_input_file = df_input_file.fillna(df_input_file.mean()["one":"two"])
print("replace nan values with the mean value in columns one and two:")
print(df_input_file)
print()

df_input_file = df_input_file.fillna(df_input_file.median()[:"three"])
print("replace nan values with the median value in column three:")
print(df_input_file)
print()

df_input_file["four"] = df_input_file["four"].fillna("club")
print("replace nan values with the 'club' string in column four:")
print(df_input_file)
print()

df_input_file["five"] = df_input_file["five"].fillna(True)
print("replace nan values with the true boolean in column five:")
print(df_input_file)
print()

now = datetime.datetime.now().strftime("%m/%d/%Y")
df_input_file["six"] = df_input_file["six"].fillna(str(now))
df_input_file["six"] = pd.to_datetime(df_input_file["six"])
print("replace nan values with the current date in column six:")
print(df_input_file)
print()

df_input_file["seven"] = df_input_file["seven"].fillna("£0.0")
```

```
    print("replace nan values with the £0.0 in column seven:")
    print(df_input_file)
    print()

    df_input_file["seven"] = df_input_file["seven"].map(lambda x: str(x)[1:])
    print("remove the first character (English pound symbol) in column seven:")
    print(df_input_file)
    print()

    df_input_file["eight"].replace
          (to_replace=dict(BMC="BioMed Central",
          ACS="American Chemical Society",
          BPS="Biophysical Society",
          CJS="Cadmus Journal Services",
          FM="Frontiers Media"), inplace=True)
    print("substitute the abbreviation in column eight:")
    print(df_input_file)
    print()

    x = df_input_file.drop(labels="eight", axis=1)
    print("x: labels by dropping column eight:")
    print(x)
    print()

    x = df_input_file.iloc[:, 0:6].values
    print("x: labels selection by using index location method ilo():")
    print(x)
    print()

    y = df_input_file["eight"]
    print("y: target selection by column eight:")
    print(y)
    print()

    y = df_input_file.iloc[:, 7].values
    print("y: target selection by using index location method ilo():")
    print(y)
    print()

    df_input_file.to_csv(output_file_path, encoding="latin1")
    print("the output file has been created successfully:")
    print(output_file_path)
    print()

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

The config.py module includes the input and output CSV files name.

```
INPUT_FILE_PATH = "input_data.csv"
OUTPUT_FILE_PATH = "output_data.csv"
```

Results:

```
using pandas library for data manipulation and cleansing:

project directory path:
C:\iubh_fernstudium\src\4.2_scientific_calculations

input file path:
C:\iubh_fernstudium\src\4.2_scientific_calculations\input_data.csv

output file path:
C:\iubh_fernstudium\src\4.2_scientific_calculations\output_data.csv

pandas data frame for input file:
   one two three  four  five      six   seven eight
0 NaN 2.1   3.1   bar  True      NaN £200.45  BMC
1 NaN NaN   3.2   NaN False 1/1/2017 £650.38  ACS
2 3.0 2.3   3.3  club  True 2/1/2017  £85.07  ACS
3 5.0 2.4   NaN   bar   NaN 3/1/2017     NaN  BPS
4 NaN 2.5   3.5   bar False      NaN £230.22  BMC
5 6.0 NaN   3.3  club  True 2/1/2017 £650.96  CJS
6 2.0 2.5   NaN house   NaN 4/1/2017     NaN   FM
7 3.0 2.3   3.3  club  True 2/1/2017  £85.07  ACS

number of rows and columns:
(8, 8)

file information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8 entries, 0 to 7
Data columns (total 8 columns):
one     5 non-null float64
two     6 non-null float64
three   6 non-null float64
four    7 non-null object
five    6 non-null object
six     6 non-null object
seven   6 non-null object
eight   8 non-null object
dtypes: float64(3), object(5)
memory usage: 640.0+ bytes

summary descriptive stats for numerical columns:
            one      two    three
count 5.000000 6.000000 6.000000
```

```
mean  3.800000 2.350000 3.283333
std   1.643168 0.151658 0.132916
min   2.000000 2.100000 3.100000
25%   3.000000 2.300000 3.225000
50%   3.000000 2.350000 3.300000
75%   5.000000 2.475000 3.300000
max   6.000000 2.500000 3.500000

frequency distribution for categorical column four:
club    3
bar     3
house   1
Name: four, dtype: int64

frequency distribution for categorical column eight:
ACS   3
BMC   2
CJS   1
FM    1
BPS   1
Name: eight, dtype: int64

missing values (nan) by columns:
one     3
two     2
three   2
four    1
five    2
six     2
seven   2
eight   0
dtype: int64

duplicated rows:
0   False
1   False
2   False
3   False
4   False
5   False
6   False
7    True
dtype: bool

drop duplicated rows and keep the first:
  one two three  four  five      six   seven eight
0 NaN 2.1   3.1   bar  True      NaN £200.45  BMC
1 NaN NaN   3.2   NaN False 1/1/2017 £650.38  ACS
```

```
2 3.0 2.3   3.3  club  True 2/1/2017  £85.07  ACS
3 5.0 2.4   NaN   bar   NaN 3/1/2017     NaN  BPS
4 NaN 2.5   3.5   bar False      NaN £230.22  BMC
5 6.0 NaN   3.3  club  True 2/1/2017 £650.96  CJS
6 2.0 2.5   NaN house   NaN 4/1/2017     NaN   FM

replace nan values with the mean value in columns one and two:
  one  two three  four  five      six   seven eight
0 4.0 2.10   3.1   bar  True      NaN £200.45  BMC
1 4.0 2.36   3.2   NaN False 1/1/2017 £650.38  ACS
2 3.0 2.30   3.3  club  True 2/1/2017  £85.07  ACS
3 5.0 2.40   NaN   bar   NaN 3/1/2017     NaN  BPS
4 4.0 2.50   3.5   bar False      NaN £230.22  BMC
5 6.0 2.36   3.3  club  True 2/1/2017 £650.96  CJS
6 2.0 2.50   NaN house   NaN 4/1/2017     NaN   FM

replace nan values with the median value in column three:
  one  two three  four  five      six   seven eight
0 4.0 2.10   3.1   bar  True      NaN £200.45  BMC
1 4.0 2.36   3.2   NaN False 1/1/2017 £650.38  ACS
2 3.0 2.30   3.3  club  True 2/1/2017  £85.07  ACS
3 5.0 2.40   3.3   bar   NaN 3/1/2017     NaN  BPS
4 4.0 2.50   3.5   bar False      NaN £230.22  BMC
5 6.0 2.36   3.3  club  True 2/1/2017 £650.96  CJS
6 2.0 2.50   3.3 house   NaN 4/1/2017     NaN   FM

replace nan values with the 'club' string in column four:
  one  two three  four  five      six   seven eight
0 4.0 2.10   3.1   bar  True      NaN £200.45  BMC
1 4.0 2.36   3.2  club False 1/1/2017 £650.38  ACS
2 3.0 2.30   3.3  club  True 2/1/2017  £85.07  ACS
3 5.0 2.40   3.3   bar   NaN 3/1/2017     NaN  BPS
4 4.0 2.50   3.5   bar False      NaN £230.22  BMC
5 6.0 2.36   3.3  club  True 2/1/2017 £650.96  CJS
6 2.0 2.50   3.3 house   NaN 4/1/2017     NaN   FM

replace nan values with the true boolean in column five:
  one  two three  four  five      six   seven eight
0 4.0 2.10   3.1   bar  True      NaN £200.45  BMC
1 4.0 2.36   3.2  club False 1/1/2017 £650.38  ACS
2 3.0 2.30   3.3  club  True 2/1/2017  £85.07  ACS
3 5.0 2.40   3.3   bar  True 3/1/2017     NaN  BPS
4 4.0 2.50   3.5   bar False      NaN £230.22  BMC
5 6.0 2.36   3.3  club  True 2/1/2017 £650.96  CJS
6 2.0 2.50   3.3 house  True 4/1/2017     NaN   FM

replace nan values with the current date in column six:
  one  two three  four  five       six   seven eight
```

```
0 4.0 2.10   3.1   bar  True 2019-10-05 £200.45  BMC
1 4.0 2.36   3.2  club False 2017-01-01 £650.38  ACS
2 3.0 2.30   3.3  club  True 2017-02-01  £85.07  ACS
3 5.0 2.40   3.3   bar  True 2017-03-01     NaN  BPS
4 4.0 2.50   3.5   bar False 2019-10-05 £230.22  BMC
5 6.0 2.36   3.3  club  True 2017-02-01 £650.96  CJS
6 2.0 2.50   3.3 house  True 2017-04-01     NaN   FM

replace nan values with the £0.0 in column seven:
  one  two three  four  five       six   seven eight
0 4.0 2.10   3.1   bar  True 2019-10-05 £200.45  BMC
1 4.0 2.36   3.2  club False 2017-01-01 £650.38  ACS
2 3.0 2.30   3.3  club  True 2017-02-01  £85.07  ACS
3 5.0 2.40   3.3   bar  True 2017-03-01    £0.0  BPS
4 4.0 2.50   3.5   bar False 2019-10-05 £230.22  BMC
5 6.0 2.36   3.3  club  True 2017-02-01 £650.96  CJS
6 2.0 2.50   3.3 house  True 2017-04-01    £0.0   FM

remove the first character (English pound symbol) in column seven:
  one  two three  four  five       six  seven eight
0 4.0 2.10   3.1   bar  True 2019-10-05 200.45  BMC
1 4.0 2.36   3.2  club False 2017-01-01 650.38  ACS
2 3.0 2.30   3.3  club  True 2017-02-01  85.07  ACS
3 5.0 2.40   3.3   bar  True 2017-03-01    0.0  BPS
4 4.0 2.50   3.5   bar False 2019-10-05 230.22  BMC
5 6.0 2.36   3.3  club  True 2017-02-01 650.96  CJS
6 2.0 2.50   3.3 house  True 2017-04-01    0.0   FM

substitute the abbreviation in column eight:
  one  two three  four  five       six  seven                     eight
0 4.0 2.10   3.1   bar  True 2019-10-05 200.45            BioMed Central
1 4.0 2.36   3.2  club False 2017-01-01 650.38 American Chemical Society
2 3.0 2.30   3.3  club  True 2017-02-01  85.07 American Chemical Society
3 5.0 2.40   3.3   bar  True 2017-03-01    0.0       Biophysical Society
4 4.0 2.50   3.5   bar False 2019-10-05 230.22            BioMed Central
5 6.0 2.36   3.3  club  True 2017-02-01 650.96   Cadmus Journal Services
6 2.0 2.50   3.3 house  True 2017-04-01    0.0           Frontiers Media

x: labels by dropping column eight:
  one  two three  four  five       six  seven
0 4.0 2.10   3.1   bar  True 2019-10-05 200.45
1 4.0 2.36   3.2  club False 2017-01-01 650.38
2 3.0 2.30   3.3  club  True 2017-02-01  85.07
3 5.0 2.40   3.3   bar  True 2017-03-01    0.0
4 4.0 2.50   3.5   bar False 2019-10-05 230.22
5 6.0 2.36   3.3  club  True 2017-02-01 650.96
6 2.0 2.50   3.3 house  True 2017-04-01    0.0
```

```
x: labels selection by using index location method ilo():
[[4.0 2.1 3.1 'bar' True Timestamp('2019-10-05 00:00:00')]
 [4.0 2.3600000000000003 3.2 'club' False
 Timestamp('2017-01-01 00:00:00')]
 [3.0 2.3 3.3 'club' True Timestamp('2017-02-01 00:00:00')]
 [5.0 2.4 3.3 'bar' True Timestamp('2017-03-01 00:00:00')]
 [4.0 2.5 3.5 'bar' False Timestamp('2019-10-05 00:00:00')]
 [6.0 2.3600000000000003 3.3 'club' True Timestamp('2017-02-01 00:00:00')]
 [2.0 2.5 3.3 'house' True Timestamp('2017-04-01 00:00:00')]]

y: target selection by column eight:
0   BioMed Central
1   American Chemical Society
2   American Chemical Society
3   Biophysical Society
4   BioMed Central
5   Cadmus Journal Services
6   Frontiers Media
Name: eight, dtype: object

y: target selection by using index location method ilo():
['BioMed Central' 'American Chemical Society' 'American Chemical Society'
 'Biophysical Society' 'BioMed Central' 'Cadmus Journal Services'
 'Frontiers Media']

the output file has been created successfully:
C:\iubh_fernstudium\src\4.2_scientific_calculations\output_data.csv
```

SciPy Library

SciPy is well-known as the library of scientific algorithms for Python. It builds on the low-level NumPy library for one-dimensional and multidimensional arrays. This library provides a large number of higher-level scientific algorithms in Python. Some of the main sublibraries are

- special functions,
- integration,
- optimization,
- interpolation,
- fourier transforms,
- signal processing,
- linear algebra,
- sparse eigenvalue problems,
- statistics,
- multi-dimensional image processing, and
- file IO.

Each of these submodules provides a number of functions and classes that can be used to solve problems in their respective topics.

We will now look at some code examples involving special functions to find out how SciPy library is used. The example below will show you how to calculate the first and second Bessel functions, the definite integral, and the linear and cubic functions interpolation.

Examples:

```
import numpy as np
from scipy.special import jn, yn, jn_zeros, yn_zeros
from scipy.integrate import quad, dblquad, tplquad
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
import random as rand

def function_defined(x):
   return np.cos(x)

def main():
  # example 1: calculate first and second kind bessel functions
   alpha = 0
   x = 0.0
   print("first kind bessel function")
   print("J_{}({}) = {}".format(alpha, x, jn(alpha, x)))
   x = 1.0
   print("second kind bessel function")
   print("Y_{}({}) = {}".format(alpha, x, yn(alpha, x)))
  # plot four bessel functions
   x = np.linspace(0, 10, 100)
   fig, ax = plt.subplots()
   for alpha in range(4):
       ax.plot(x, jn(alpha, x), label=r"J$_{}(x)$".format(alpha))
   ax.legend();
   plt.xlabel("x")
   plt.ylabel("y")
   plt.title("Bessel Functions")
   plt.show()

  # example 2: calculate definite integral
   a = 0
   b = 1
   integral_value, absolute_error = quad(function_defined, a, b)
   print("integral value: {} \nabsolute error:
 {}".format(integral_value, absolute_error))

  # example 3. linear and cubic functions interpolation
   n = np.arange(0, 10)
```

```
    x = np.linspace(0, 9, 100)
   # set the actual function
    y_actual = function_defined(x)
    # generate random experiment data
    y_experiment = function_defined(n) + 0.1 * np.random.randn(len(n))
    linear_interpolation = interp1d(n, y_experiment, kind="linear")
    # get linear interpolation
    y_linear_interpolation = linear_interpolation(x)
    cubic_interpolation = interp1d(n, y_experiment, kind="cubic")
   # get cubic interpolation
    y_cubic_interpolation = cubic_interpolation(x)
   # plot y actual, experiment, linear and cubic interpolation
    fig, ax = plt.subplots(figsize=(10,4))
    ax.plot(n, y_experiment, "bs", label="experiment data")
    ax.plot(x, y_actual, "k", lw=2, label="actual function")
    ax.plot(x, y_linear_interpolation, "r", label="linear interpolation")
    ax.plot(x, y_cubic_interpolation, "g", label="cubic interpolation")
    ax.legend(loc=3);
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Functions Interpolation Example")
    plt.show()

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

Results:

first kind bessel function

J_0(0.0) = 1.0

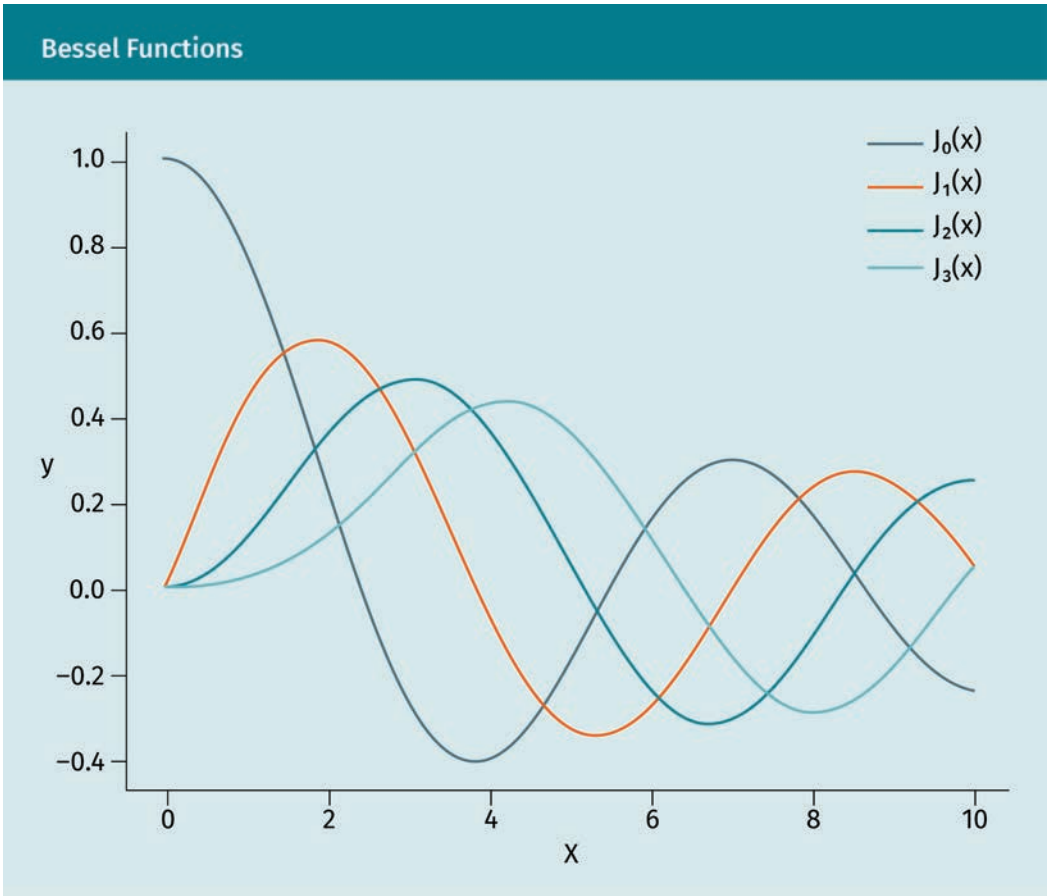
second kind bessel function

Y_0(1.0) = 0.08825696421567697

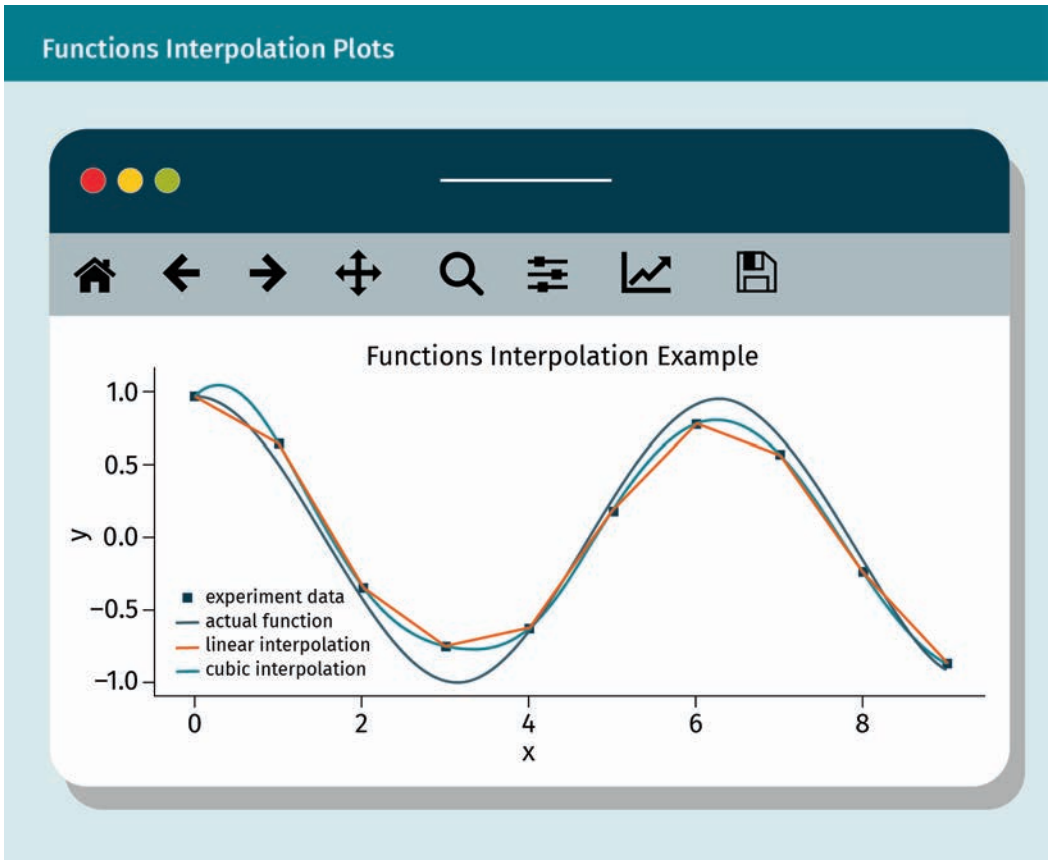
integral value: 0.8414709848078965

absolute error: 9.34220461887732e-15

The figure below shows the Bessel function for $\alpha = 0, 1, 2, 3$.



The following figure shows a comparison graph between experiment data, actual function, and linear and cubic interpolation.



4.3 Speed Up Python

As we know already, Python is an interpreted programming language. The Python programs need an interpreter to covert the code into generic machine code. Each time a Python module runs, the code needs to be parsed and interpreted before execution. A number of comparison studies state that Python is much slower than a number of compiled programming languages such as C++, Java, C#, etc.

Speeding up Python programs is a very important task for software engineers today, especially in machine learning projects with large datasets and multidimensional correlated input parameters (features). Below are the most popular tools that are used to speed up Python programs:

- Cython is a programming language designed to give C-like performance with code that is written mostly in Python with optional additional C-inspired syntax. Cython is a compiled language that is typically used to generate CPython extension modules.
- PyPy is built using the RPython language that compiles the Python code into efficient code as statically-typed language. At this point, the PyPy then translates the generated RPython code into a form of bytecode, together with an interpreter written in the C programming language. At the end, this code is then compiled into

machine code and the created bytecode runs on the compiled code. The package contains a "just-in-time" (JIT) compile version that you can use to speed up Python code. JIT compilation is a way of executing computer code that involves compilation during the execution of a program at run time, rather than prior to execution (Beazley & Jones, 2013).

Numba
The Numba tool offers a range of options for parallelizing Python code for CPUs and GPUs, often with only minor code changes.

- **Numba** is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code at run time. It is the most common program used to speed up Python programs today. It was specifically designed for scientific computing with NumPy arrays and functions. Numba can automatically execute NumPy array expressions on multiple CPU cores, which makes it easy to write parallel loops.

Numba

We will now look at some example code to find out how Numba can accelerate Python code. As a Python standard library, Numba can be installed using "pip install numba" or "conda install numba" with the Anaconda Distribution package. To import this library into a Python module, the line "from numba import jit" needs to be included at the top of the program.

Example:

```
import time
from numba import jit
import numpy as np

@jit(nopython=True)
def sum2dimensional(np_array):
   array_rows, array_columns = np_array.shape
   result = 0.0
   for i in range(array_rows):
       for j in range(array_columns):
           result += np_array[i,j]
   return result

def main():
   np_array = np.arange(100000000).reshape(10000, 10000)
   print("sum calculation: {}".format(sum2dimensional(np_array)))

if __name__ == '__main__':
   # method of time module in Python is used to get the current
processor time as a floating point number expressed in seconds
   start_time = time.clock()
   main()
   end_time = time.clock()
   diff_time = end_time - start_time
   result = time.strftime("%H:%M:%S", time.gmtime(diff_time))
   print("program runtime: {}".format(result))
```

With "`@jit(nopython=True)`" decorator, the following result occurs:

Result:

```
sum calculation: 4999999950000000.0

program runtime: 00:00:00
```

Without "`@jit(nopython=True)`" decorator, the result is:

Result:

```
sum calculation: 4999999950000000.0

program runtime: 00:00:28
```

As you can see, there was no runtime with the Numba decorator "`@jit(nopython=True)`". Without the decorator, the runtime was 28 seconds. The "nopython" compilation mode was used. The result of these runtime program comparisons may vary depending on the hardware machine that is used.

4.4 Data Visualization

Data visualization is an important step in any data statistical analysis or machine learning project (McKinney, 2017). This visualization is generally presented using maps or graphs. It helps to see and understand the data so patterns, trends, and outliers can be detected early. Below are the three most used data visualization ecosystem libraries in Python:

1. Matplotlib
2. Seaborn
3. Bokeh

Matplotlib Library

The Matplotlib library is the most widely-used library for plotting in the Python community today. This library supports a wide range of graphs including scatter plots, bar charts, histograms, line plots, pie charts, stem plots, contour plots, quiver plots, and spectrograms (Matthes, 2019). A Matplotlib figure can be categorized into several parts as described below:

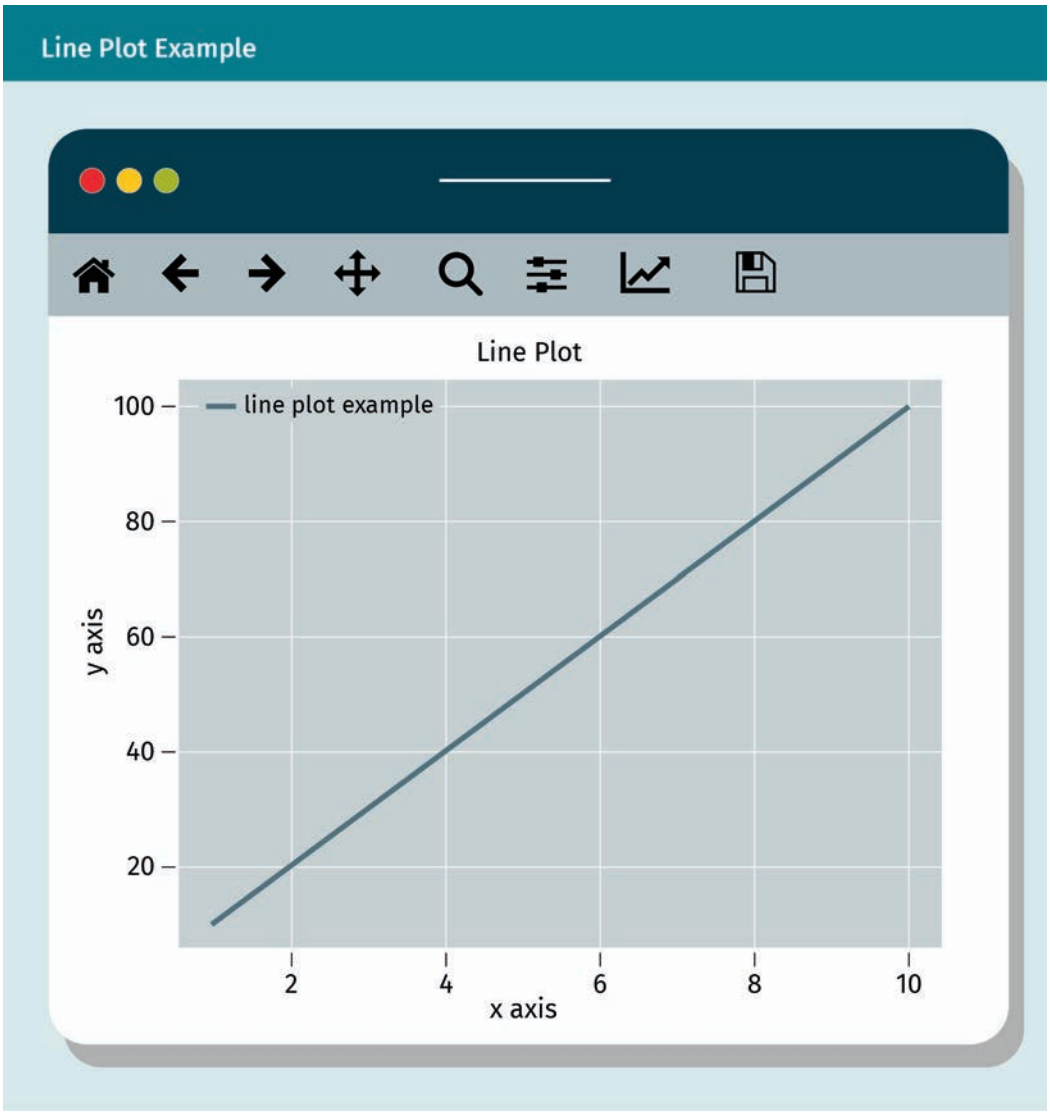
- Figure — This is a whole figure that may contain one or more axes (plots). You can think of a figure as a canvas that contains plots.
- Axes — This is what we generally think of as a plot. A figure can contain many axes. It contains two or three (in the case of 3D) axis objects. Each axes has a title, an x-label, and a y-label.
- Axis — They are the number line like objects that generate the graph limits.
- Artist — Everything that can be seen on the figure is an artist. Examples include text objects, Line2D objects, and collection objects.

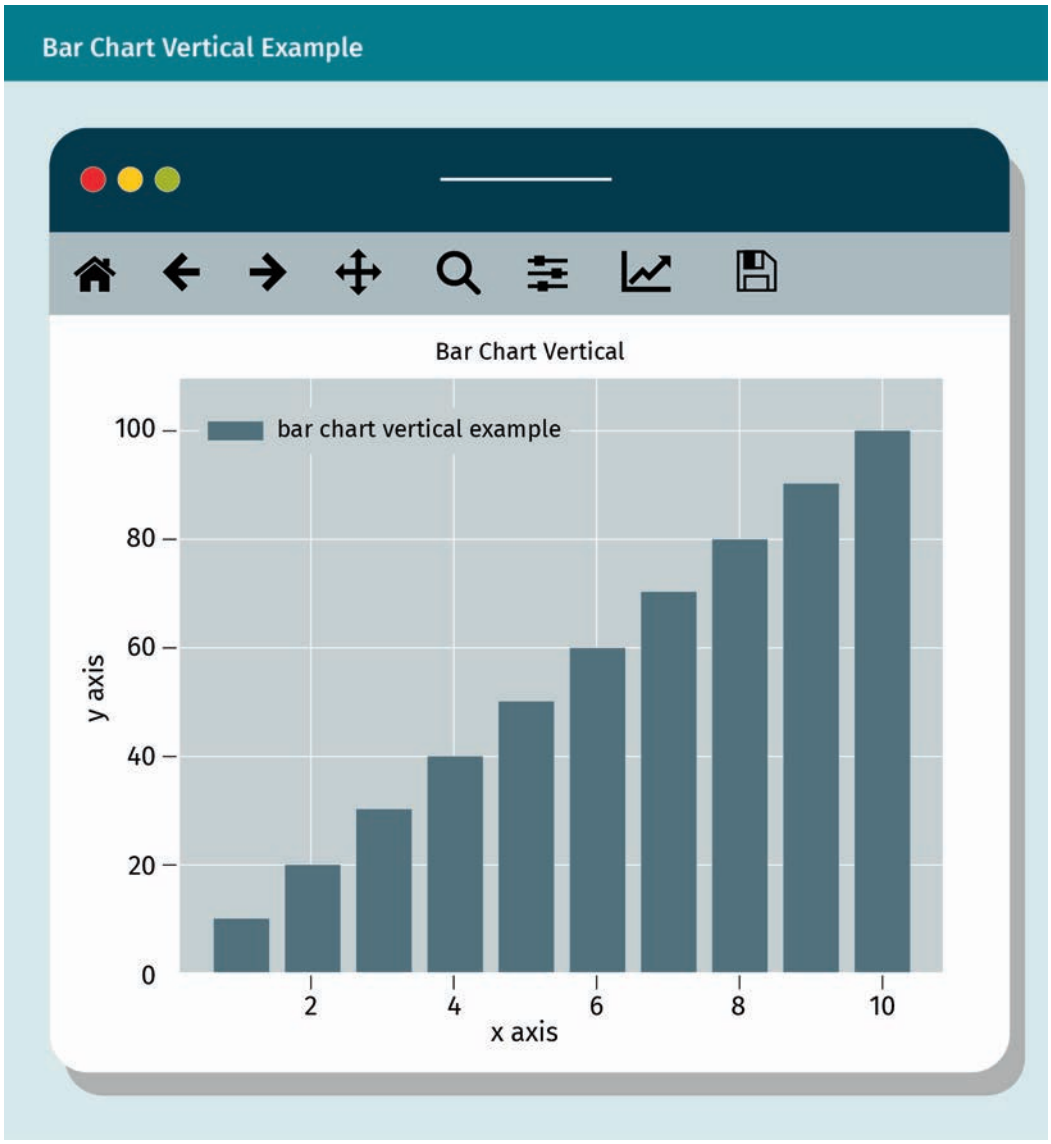
Examples:

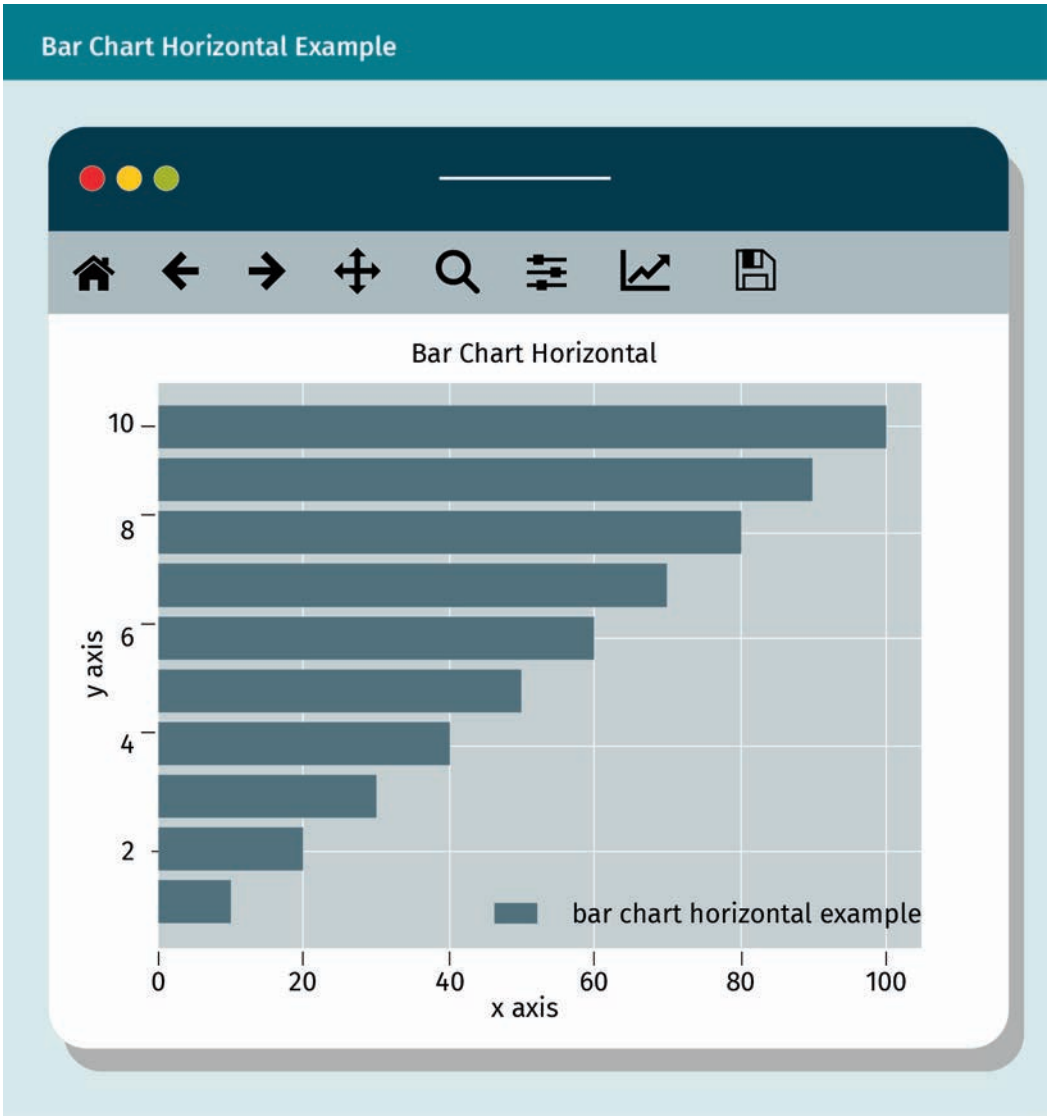
```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import style

def main():
  # x and y data list
   x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
   y = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
  # set plot style
   style.use('ggplot')
  # line plot example
   plt.plot(x, y, label="line plot example", linewidth=2)
   plt.legend()
   plt.grid(True,color="k")
   plt.ylabel('y axis')
   plt.xlabel('x axis')
   plt.title('Line Plot')
   plt.show()

  # bar chart vertical example
   plt.bar(x, y, label="bar chart vertical example", align='center')
   plt.legend()
   plt.grid(True,color="k")
   plt.ylabel('y axis')
   plt.xlabel('x axis')
   plt.title('Bar Chart Vertical')
   plt.show()

  # bar chart horizontal example
   plt.barh(x, y, label="bar chart horizontal example", align='center')
   plt.legend()
   plt.grid(True,color="k")
   plt.ylabel('y axis')
   plt.xlabel('x axis')
   plt.title('Bar Chart Horizontal')
   plt.show()

  # scatter plot example
```

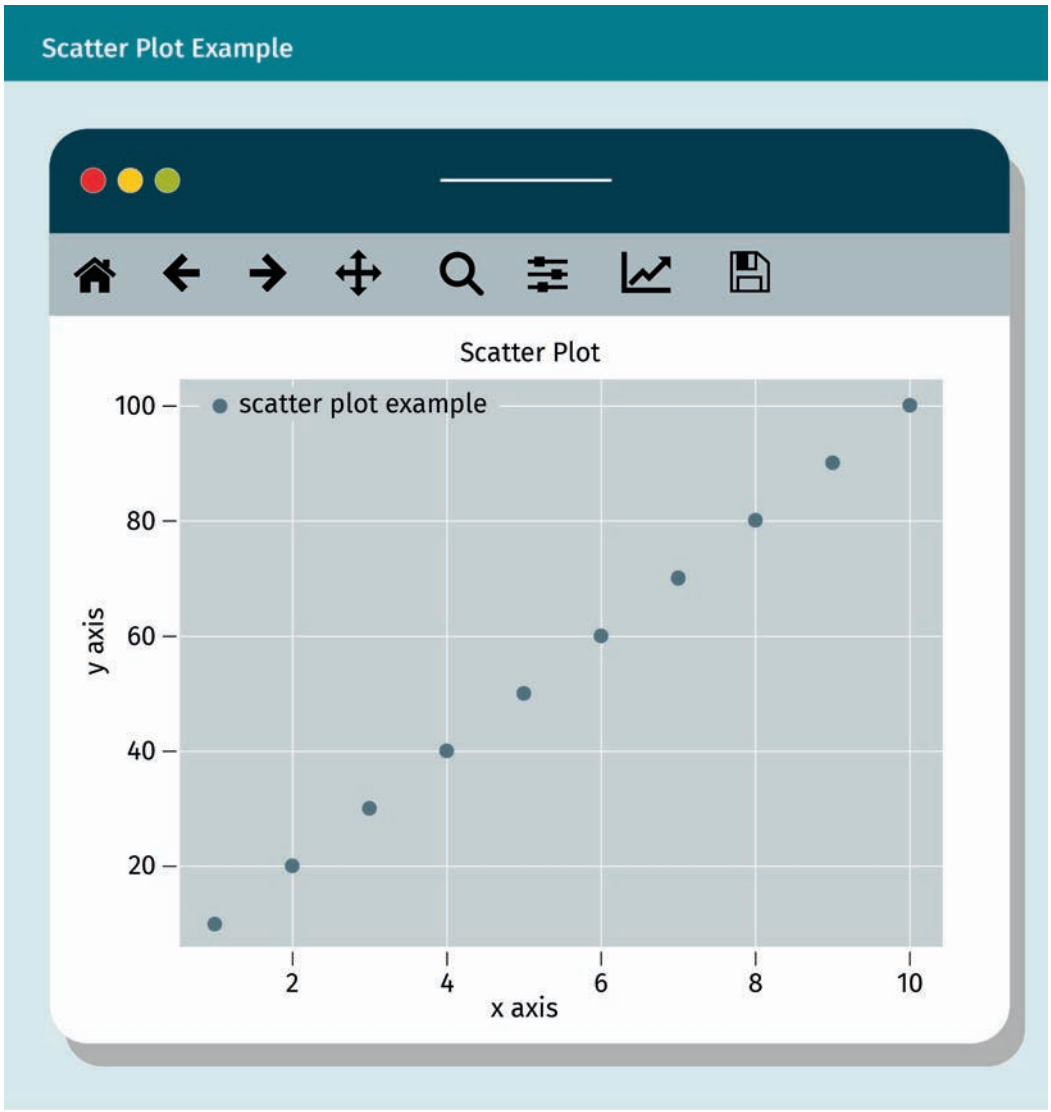
```
   plt.scatter(x, y, label="scatter plot example")
   plt.legend()
   plt.grid(True,color="k")
   plt.ylabel('y axis')
   plt.xlabel('x axis')
   plt.title('Scatter Plot')
   plt.show()

  # histogram plot example
   mu, sigma = 100, 15
   x = mu + sigma * np.random.randn(10000)
   n, bins, patches = plt.hist(x, 50, density=1,    facecolor='r', alpha=0.75,
label="histogram plot example")
   plt.text(45, 0.028, r'$\mu=100,\ \sigma=15$')
   plt.axis([40, 160, 0, 0.03])
   plt.legend()
   plt.grid(True,color="k")
   plt.xlabel("x")
   plt.ylabel("P(x)")
   plt.title("Histogram Plot")
   plt.show()

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

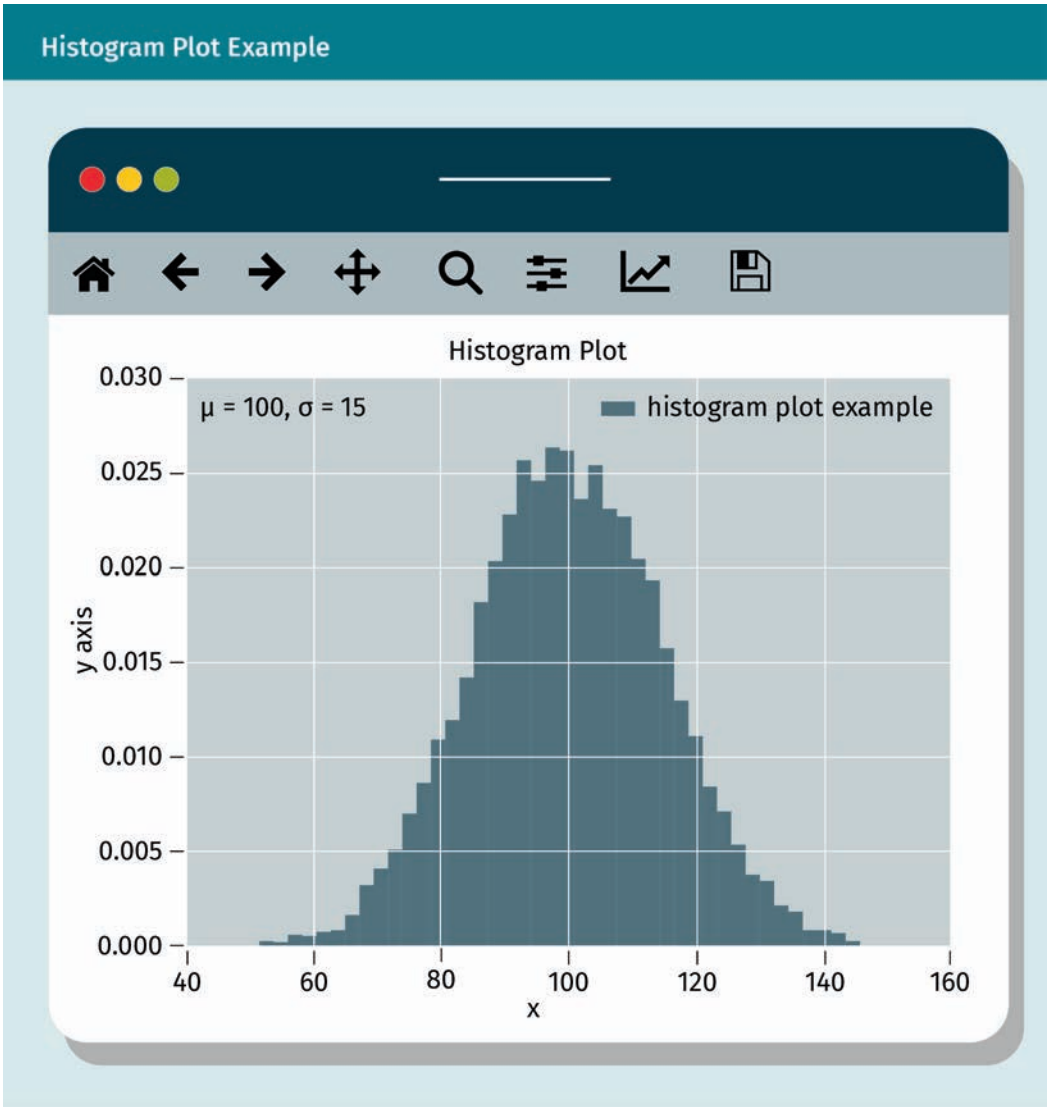
Results:


Line Plot Example
Line Plot
line plot example
100
80
60
40
20
y axis
2
4
6
8
10
x axis


Bar Chart Vertical Example
Bar Chart Vertical
bar chart vertical example
100
80
60
40
20
0
y axis
2
4
6
8
10
x axis


Bar Chart Horizontal Example
Bar Chart Horizontal
10
8
6
4
2
y axis
0
20
40
60
80
100
x axis
bar chart horizontal example


Scatter Plot Example
Scatter Plot
scatter plot example
100
80
60
40
20
y axis
2
4
6
8
10
x axis



Seaborn Library

Seaborn library builds on matplotlib library, however, this library is a complement to matplotlib, not a replacement. It was specifically developed for statistical data visualization. Seaborn provides the following features for data visualization: Develops themes for styling matplotlib graphics, visualizes univariate and bivariate array data, fits in and visualizes linear and non-linear regression models, excellent plotting for statistical time series data, built to work effectively with NumPy and pandas data structures, provides new themes for styling matplotlib plots, etc. Seaborn can be installed using “pip install seaborn” or “conda install seaborn” with the Anaconda Distribution package.

Before looking at some example plots, we will examine how seaborn can get data using the "`load_dataset()`" built-in function. The seaborn-data GitHub public repository supports different types of datasets in CSV file format. The "`load_dataset()`" function can load a dataset from this online repository, meaning that an internet connection is required in order to use this function. The example code below shows how to load the iris flowers dataset.

```
import matplotlib.pyplot as plt
import seaborn as sns

def main():
   iris_data = sns.load_dataset("iris")
   print(iris_data)

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

Results:

Iris Flowers Dataset Results				
sepal_length	sepal_width	petal_length	petal_width	species
5.1	3.5	1.4	0.2	setosa
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5	3.6	1.4	0.2	setosa
...	..	..		...
7	3.2	4.7	1.4	versicolor
6.4	3.2	4.5	1.5	versicolor
6.9	3.1	4.9	1.5	versicolor
5.5	2.3	4	1.3	versicolor

Iris Flowers Dataset Results				
6.5	2.8	4.6	1.5	versicolor
...	...	...		...
6.3	3.3	6	2.5	virginica
5.8	2.7	5.1	1.9	virginica
7.1	3	5.9	2.1	virginica
6.3	2.9	5.6	1.8	virginica
6.5	3	5.8	2.2	virginica

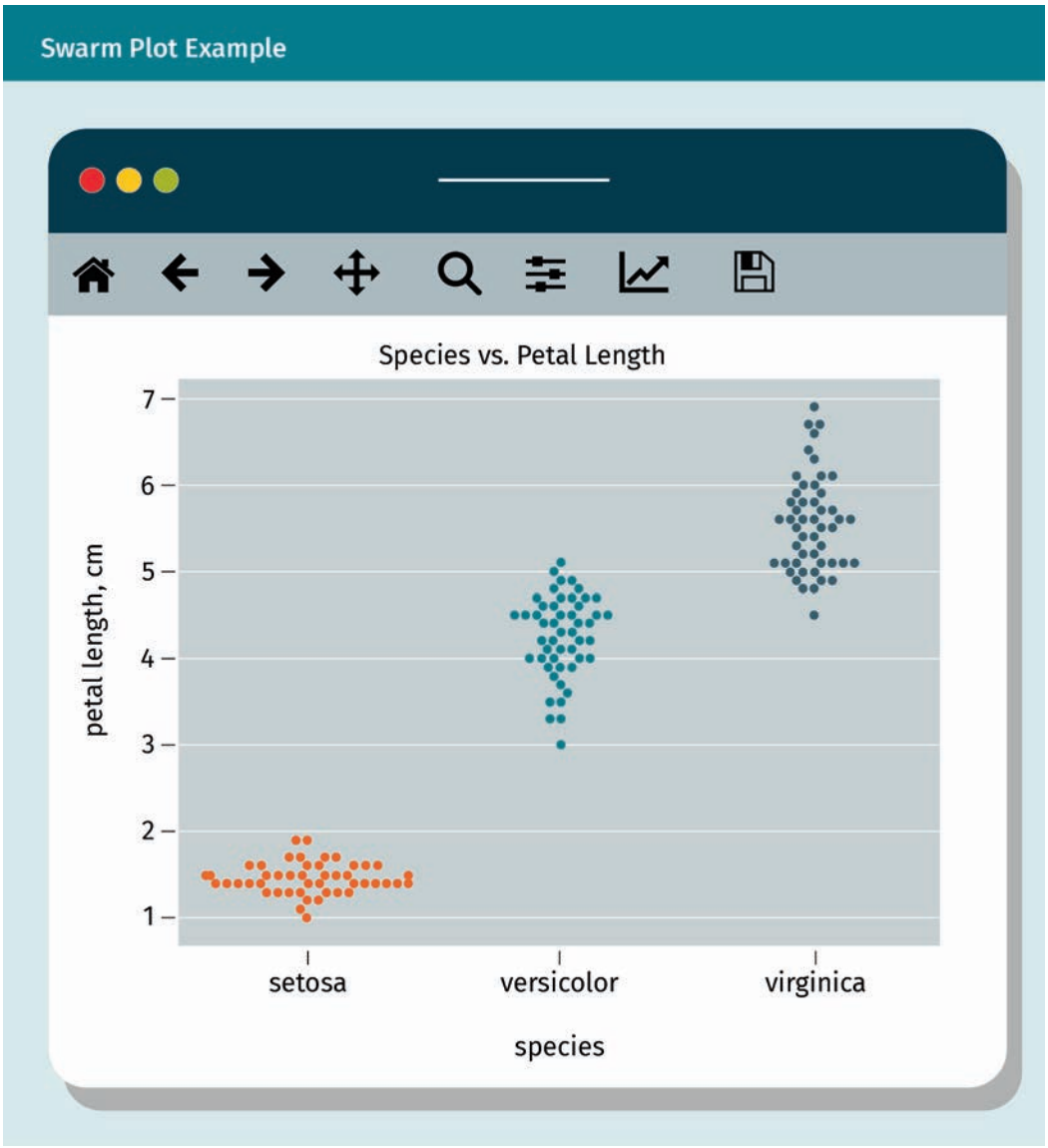
The iris flowers dataset contains three classes of species. These are setosa, versicolor, and virginica. The flowers' parameters (features) are measured in cm and they are `sepal_length`, `sepal_width`, `petal_length`, and `petal_width`. This dataset is very popular for testing new machine learning data classification models as well as being used for teaching and presentation purposes. In real production data analysis projects, the selected dataset could be loaded using the pandas `read_csv()` method that was explained earlier.

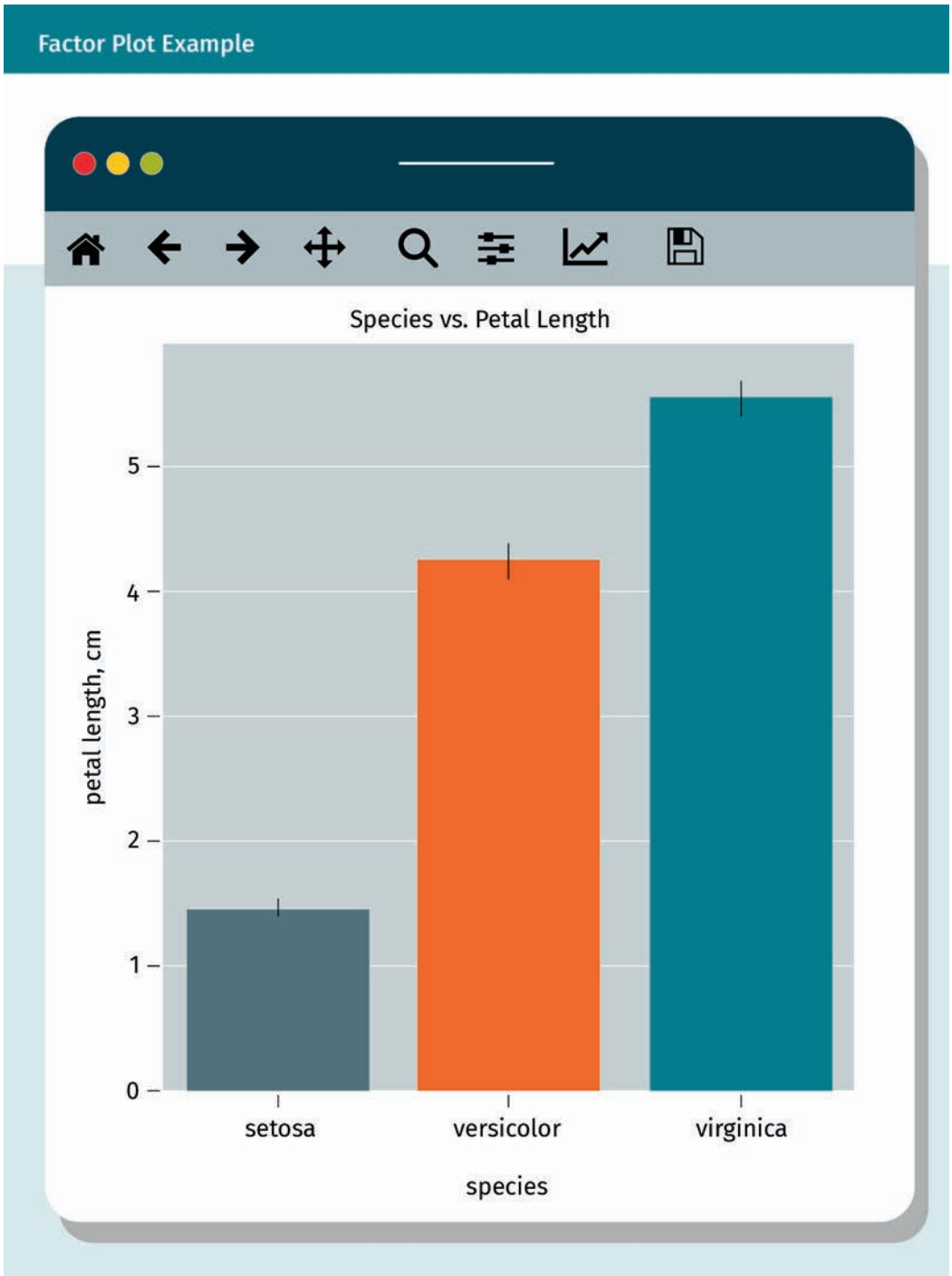
We will now build some seaborn plots using this dataset.

```
import matplotlib.pyplot as plt
from matplotlib import style
import seaborn as sns

def main():
   iris_data = sns.load_dataset(name="iris")
# print(iris_data)
   style.use('ggplot')
# swarm plot example
   sns.swarmplot(x="species", y="petal_length", data=iris_data)
   plt.text(40, 7, "abc")
   plt.ylabel("petal length, cm")
   plt.xlabel("species")
   plt.title("Species vs. Petal Length")
   plt.show()
# factor plot example
   g = sns.factorplot("species", "petal_length", data=iris_data, kind="bar",
palette="muted", legend=False)
   plt.ylabel("petal length, cm")
```

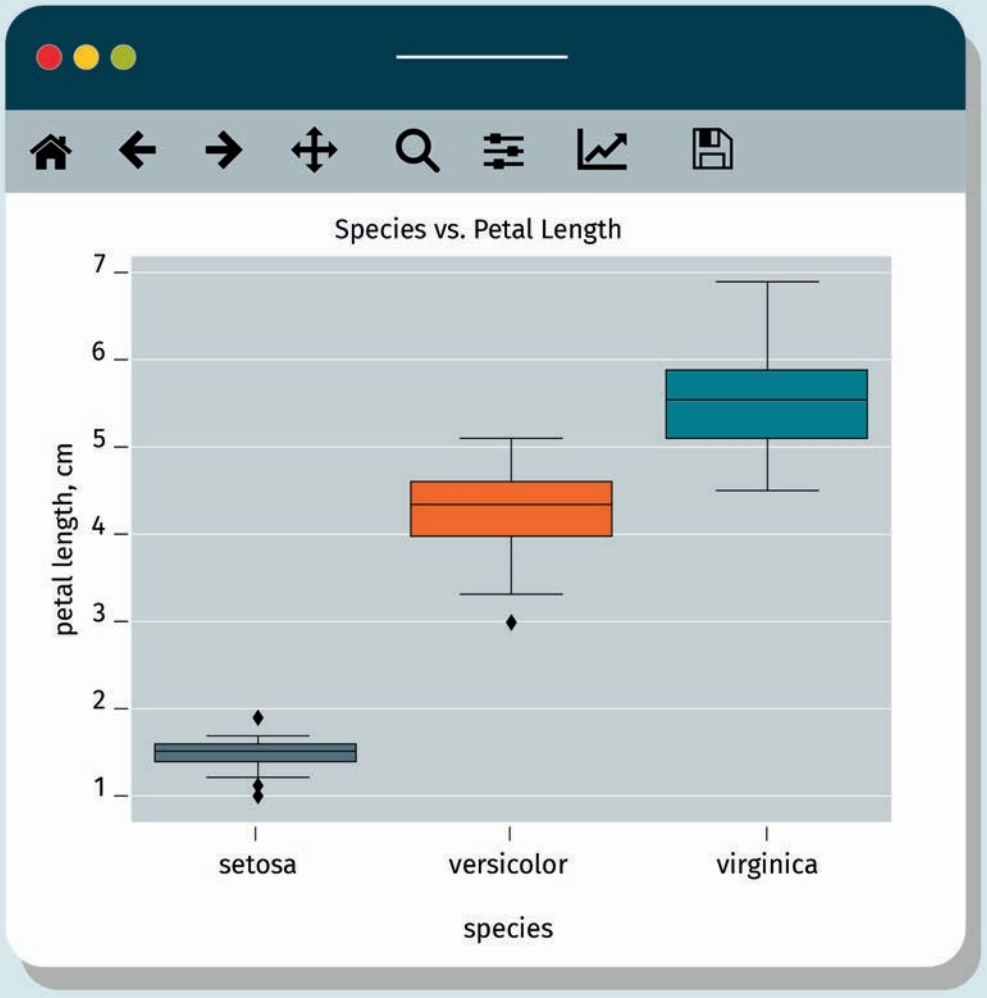
```
    plt.xlabel("species")
    plt.title("Species vs. Petal Length")
    plt.show()
# box plot example
    sns.boxplot(x="species", y="petal_length", data=iris_data)
    plt.ylabel("petal length, cm")
    plt.xlabel("species")
    plt.title("Species vs. Petal Length")
    plt.show()
# pair plot example
    sns.pairplot(iris_data, hue="species", size=2)
    plt.ylabel("petal length, cm")
    plt.xlabel("species")
    plt.show()

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

Results:

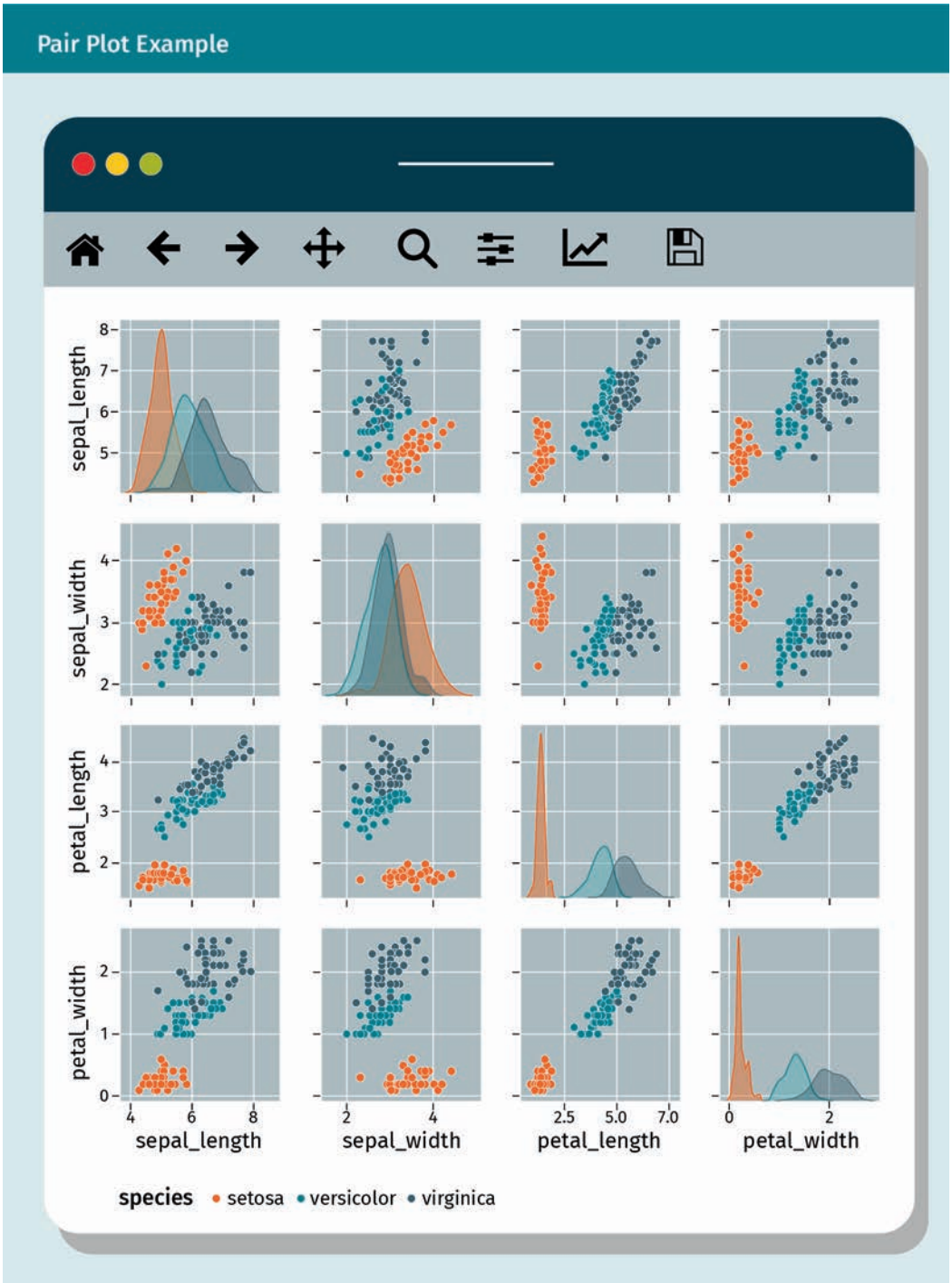

Swarm Plot Example
Species vs. Petal Length
petal length, cm
7
6
5
4
3
2
1
setosa
versicolor
virginica
species


Factor Plot Example
Species vs. Petal Length
petal length, cm
0
1
2
3
4
5
setosa
versicolor
virginica
species

Box Plot Example



Python Important Libraries



Bokeh Library

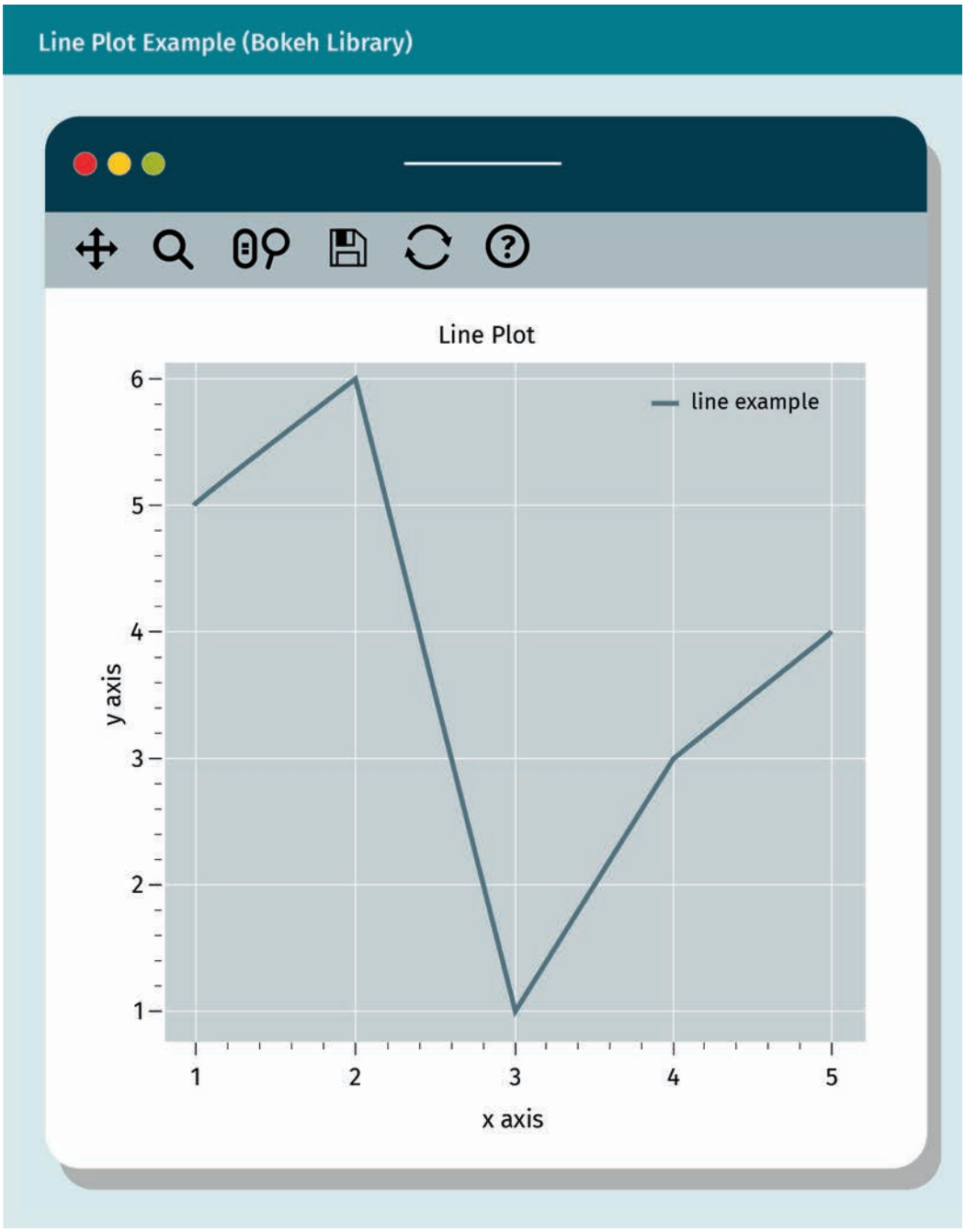
The Bokeh library is specifically used for web browsers' interactive data visualization. This library produces a JSON file, which is sent as an input to the Bokeh JavaScript library BokehJS. A JSON file stores simple data structures and objects in JavaScript Object Notation (JSON) format, which is a standard data interchange format. It is primarily used for transmitting data between a web application and a server. This client-site library BokehJS converts this JSON file into the data to be shown by any popular web browser. Bokeh can be installed using "pip install bokeh" or "conda install bokeh" with the Anaconda Distribution package. We will look at some example code to see how Bokeh creates different types of plots.

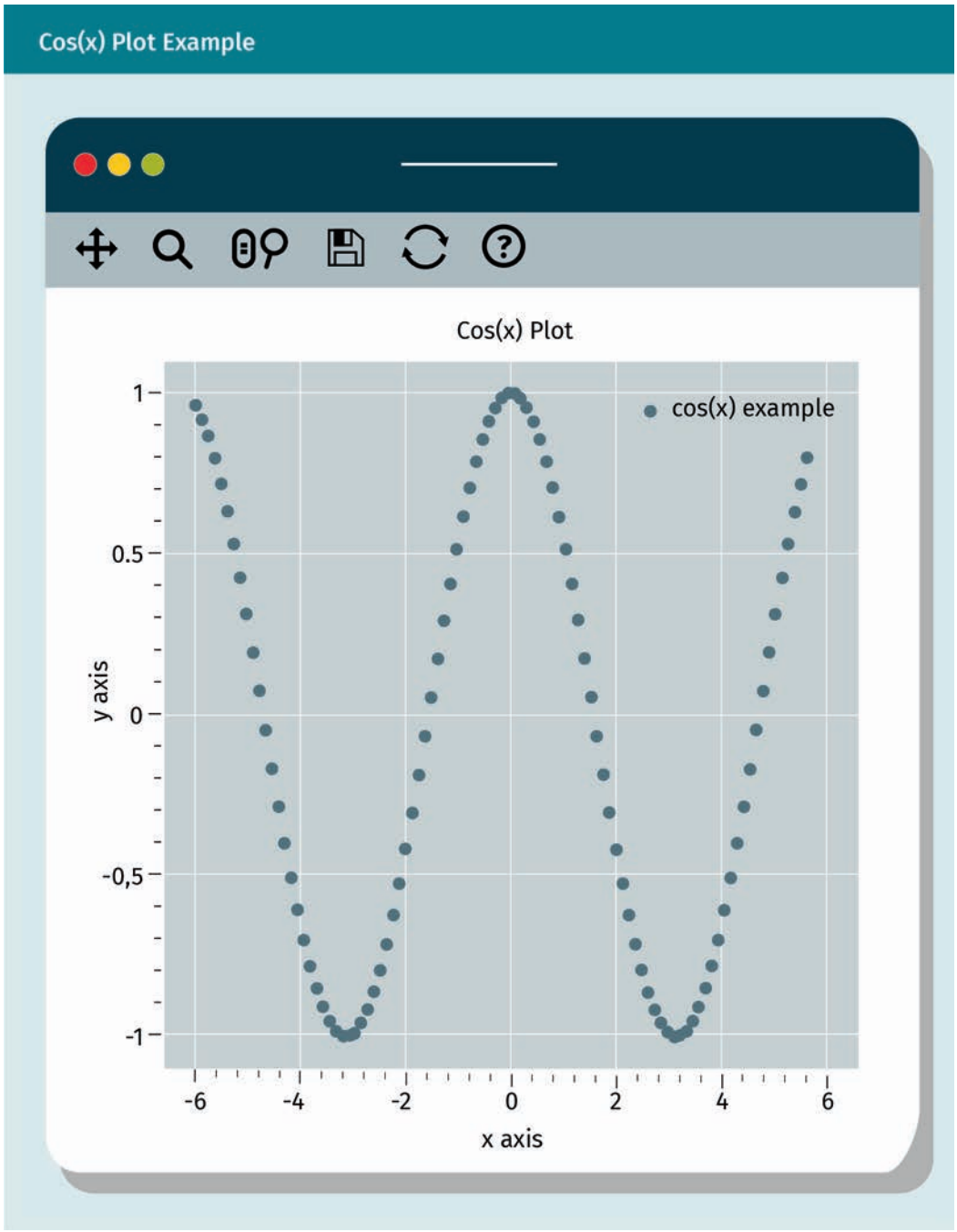
Examples:

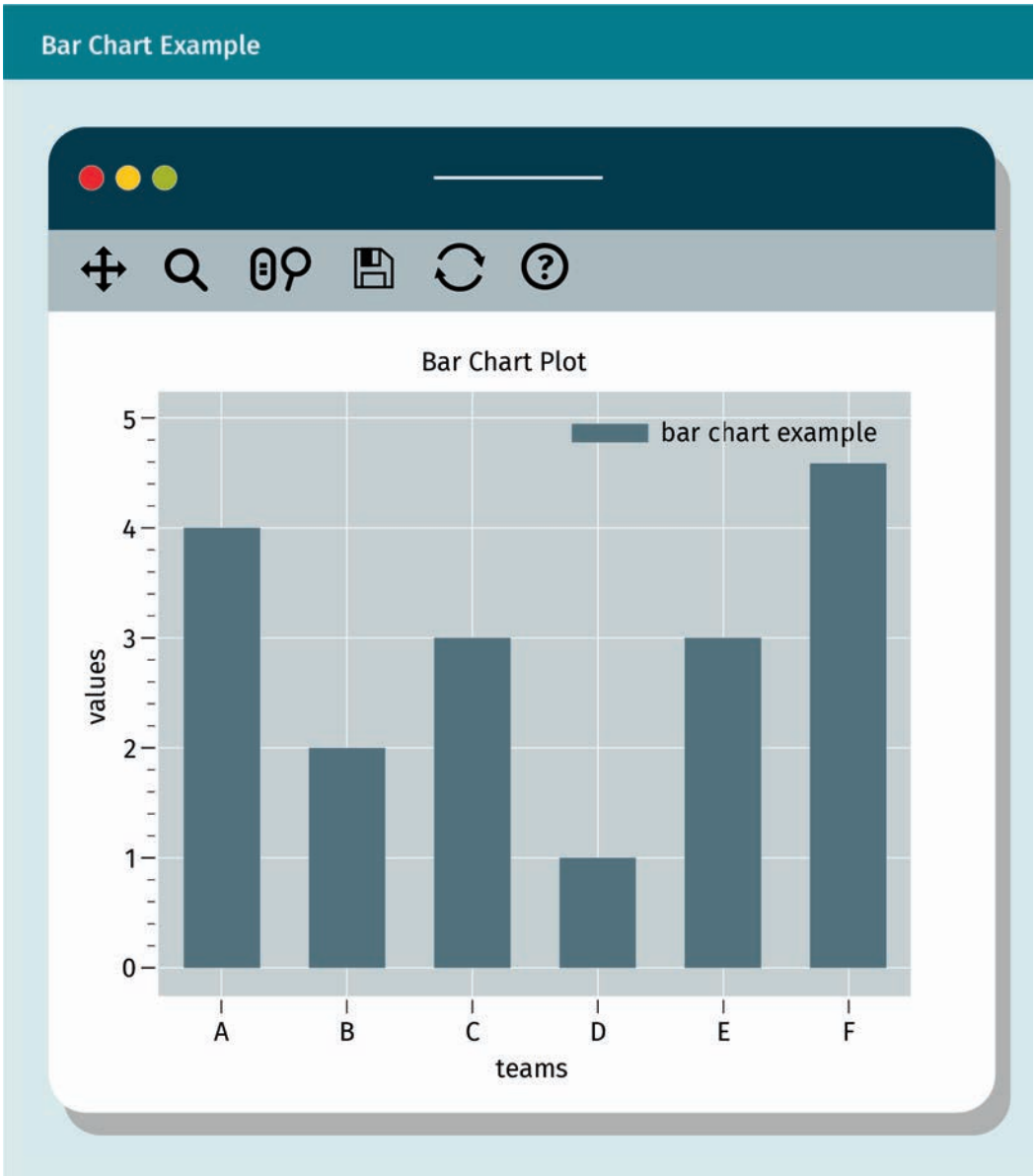
```
import numpy as np
from bokeh.plotting import figure, output_file, show
from bokeh.sampledata.iris import flowers

def main():
# line plot example
   x = [1, 2, 3, 4, 5]
   y = [5, 6, 1, 3, 4]
   output_file("line_example.html")
   plot = figure(title="Line Plot", x_axis_label="x axis",
   y_axis_label="y axis")
   plot.line(x, y, legend="line example", line_width=2)
   show(plot)
# cos(x) function example
   output_file("cosx_example.html")
   x = np.linspace(-6, 6, 100)
   y = np.cos(x)
   plot = figure(width=500, height=500, title="Cos(x) Plot",
   x_axis_label="x axis", y_axis_label="y axis")
   plot.circle(x, y, size=7, color="firebrick", alpha=0.5,
   legend="cos(x) example")
   show(plot)
#  bar chart example
   output_file("barchart_example.html")
   teams = ["A", "B", "C", "D", "E", "F"]
   plot = figure(x_range=teams, height=500, title="Bar Chart Plot",
x_axis_label="teams", y_axis_label="values")
   plot.vbar(x=teams, top=[4, 2, 3, 1, 3, 5], width=0.6,
   legend="bar chart example")
   show(plot)
# iris flowers scatter example
   output_file("iris_scatter_example.html")
   colormap = {"setosa":"orange", "versicolor":"blue", "virginica":"gray"}
   colors = [colormap[x] for x in flowers['species']]
```

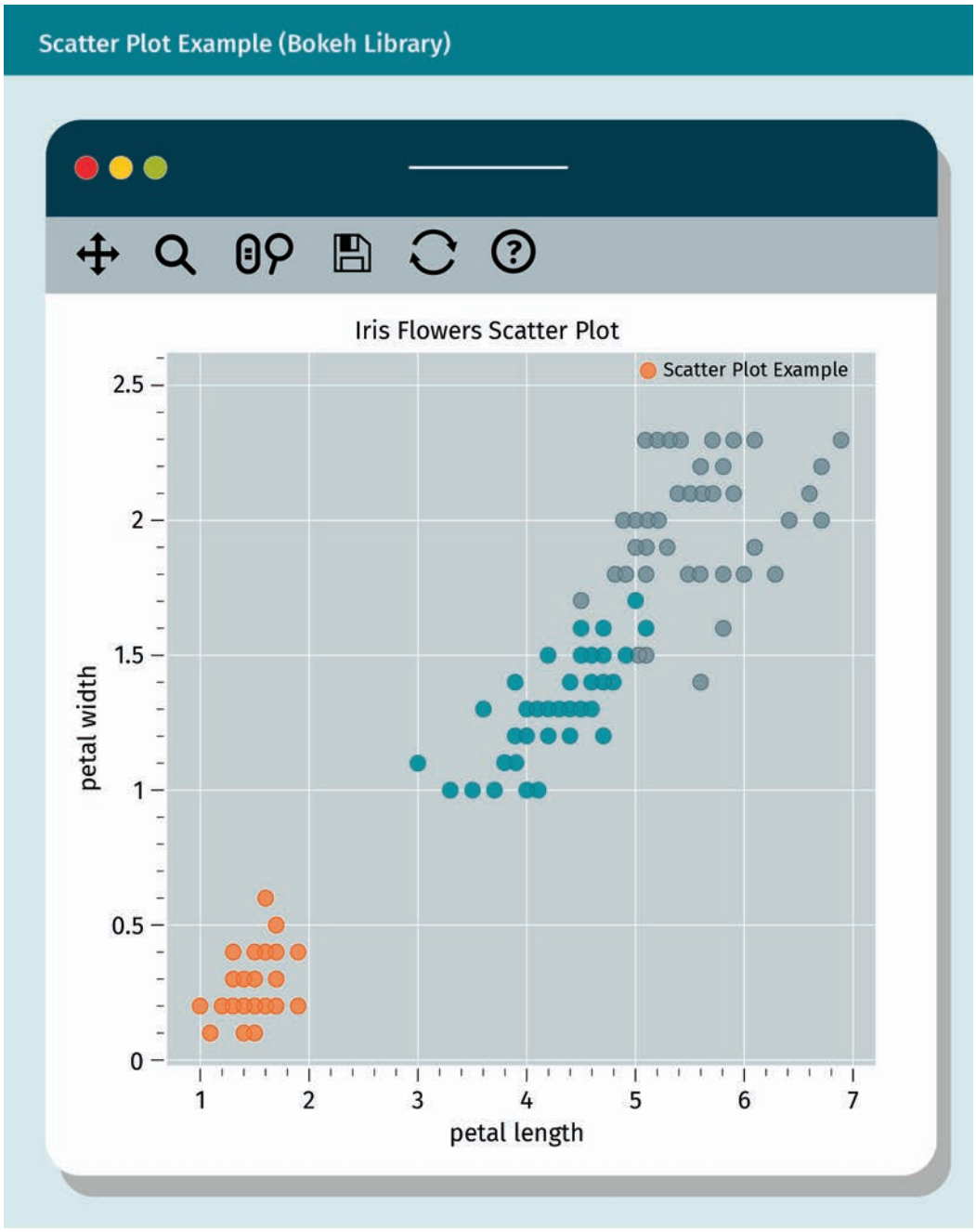
```
   plot = figure(title = "Iris Flowers Scatter Plot", x_axis_label=
   "petal length", y_axis_label="petal width")
   plot.circle(flowers["petal_length"], flowers["petal_width"],
   color=colors,
fill_alpha=0.2, size=10, legend="scatter plot example")
   show(plot)

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

Results:


Line Plot Example (Bokeh Library)
Line Plot
line example
y axis
x axis
1
2
3
4
5
6


Cos(x) Plot Example
Cos(x) Plot
cos(x) example
y axis
x axis
1
0.5
0
-0,5
-1
-6
-4
-2
0
2
4
6


Bar Chart Example
Bar Chart Plot
bar chart example
values
0
1
2
3
4
5
A
B
C
D
E
F
teams



4.5 Accessing Databases

In this section, we will examine **SQLAlchemy** — the Python SQL Toolkit (Copeland, 2008). We will demonstrate how to apply database CRUD (Create, Read, Update, Delete) operations to MySQL open source database. This library is developed in pure Python code as an open source and cross-platform framework. SQLAlchemy uses an object relational mapper (ORM) model to give application developers the full power and flexibility of

SQLAlchemy
This is an open-source SQL toolkit and object-relational mapper (ORM) for the Python programming language released under the MIT License.

SQL (structured query language). In ORM, a programming class object is mapped to a database table (entity), allowing the object model and database schema to develop a tied relationship. Consequently, this relational connection allows software engineers to considerably reduce the amount of code that needs to be written as well as the projects development/maintenance time. SQLAlchemy can be installed using "pip install sqlalchemy" or "conda install sqlalchemy" with the Anaconda Distribution package. To import SQLAlchemy in a Python module, the line "import sqlalchemy as db" needs to be added at the top of the program.

To run the SQLAlchemy code shown below, the following libraries need to be installed:

- MySQL Community Database — This is the most popular open source database engine used today. For Windows OS the MySQL installer is recommended. It is recommended to have properly setup your username and password for your local MySQL server.
- PyMySQL driver for MySQL — This is a Python MySQL client library compatible with Python Database API specification document PEP 249. PyMySQL can be installed using "pip install pymysql" or "conda install pymysql" with the Anaconda Distribution package.

We will look at the following code examples to show MySQL database CRUD operations. Make sure to create a "movie" database first using your MySQL database management interface tool.

Create Table

Now, we will define two main objects in the SQLAlchemy toolkit: Engine and connection objects.

The engine object is the starting point for any SQLAlchemy application. It is the main object for the actual database and its Python Database API Specification (DBAPI), delivered to the SQLAlchemy application through a connection pool and a dialect, which describes how to talk to a specific combination of the database and **application programming interface (API)** functions to be used. Creating an engine is just a matter of issuing a single call function "`create_engine()`".

Application programming interface
An application programming interface (API) is an interface or communication between a client and a server intended to simplify the building of client-side software.

The connection object is the interface between the database engine and the calling client application. It is the object created for the actual DBAPI connection. To create a connection object, the engine.connect() must be used. By creating the connection object, any database CRUD operations will be able to be provided to any supported database engine.

The following example will create an "actor" table with the columns `id`, `first_name`, `last_name`, `age`, `date_of_birth`, and `active`.

```
import sqlalchemy
import pymysql
import sqlalchemy as db

def main():
   # get sqlalchemy and pymysql used libraries version
   print("sqlalchemy: {}".format(sqlalchemy.__version__))
   print("pymysql: {}".format(pymysql.__version__))
  # get engine object using pymysql driver for mysql
   engine = db.create_engine("mysql+pymysql://username:password
   @localhost/movie")
  # get connection object
   connection = engine.connect()
  # get meta data object
   meta_data = db.MetaData()
  # set actor creation script table
   actor = db.Table(
   "actor", meta_data,
   db.Column("id", db.Integer, primary_key=True, autoincrement=True,
nullable=False),
   db.Column("first_name", db.String(50), nullable=False),
   db.Column("last_name", db.String(50), nullable=False),
   db.Column("age", db.Integer, nullable=False),
   db.Column("date_of_birth", db.Date, nullable=False),
   db.Column("active", db.Boolean, nullable=False))
  # create actor table and stores the information in metadata
   meta_data.create_all(engine)

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

Results:

The actor table is created. Below is the creation script table.

```
CREATE TABLE `actor` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `first_name` varchar(50) NOT NULL,
 `last_name` varchar(50) NOT NULL,
 `age` int(11) NOT NULL,
 `date_of_birth` date NOT NULL,
 `active` tinyint(1) NOT NULL,
 PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

Insert Data

The example below will add tree records into the “actor” table.

```
import sqlalchemy as db

def main():
  # get engine object using pymysql driver for mysql
   engine = db.create_engine("mysql+pymysql://username:password
   localhost/movie")
  # get metadata object
   meta_data = db.MetaData()
  # get connection object
   connection = engine.connect()
  # get actor table definition
   actor_table = db.Table("actor", meta_data, autoload=True,
   autoload_with=engine)
   # set the insert statement
   sql_query = db.insert(actor_table)
  # set data list
   data_list = [{"first_name":"John", "last_name":"Smith", "age":50,
   "date_of_birth":"1969-04-05","active":True},
                {"first_name":"Brian", "last_name":"Morgan", "age":38,
   "date_of_birth":"1981-02-11", "active":True},
                {"first_name":"David", "last_name":"White", "age":77,
   "date_of_birth":"1942-06-30", "active":False}]
  # execute the insert statement
   result = connection.execute(sql_query, data_list)

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

The following records will be inserted in to the actor table.

Records:

(1, 'John', 'Smith', 50, datetime.date(1969, 4, 5), 1)

(2, 'Brian', 'Morgan', 38, datetime.date(1981, 2, 11), 1)

(3, 'David', 'White', 77, datetime.date(1942, 6, 30), 0)

Now, we will look at the next topic and find out how to read data using the SQL Select statement.

Select Data

To select all records in the “actor” table, run the code shown below.

```
import sqlalchemy as db

def main():
  # get engine object using pymysql driver for mysql
   engine = db.create_engine("mysql+pymysql://username:password
   @localhost/movie")
   meta_data = db.MetaData()
  # get connection object
   connection = engine.connect()
  # get actor table definition
   actor_table = db.Table("actor", meta_data, autoload=True,
   autoload_with=engine)
  # set the select statement
   select_actor = db.select([actor_table])
  # execute the select statement
   dataset = connection.execute(select_actor).fetchall()
  # print row by row
   for row in dataset:
       print(row)

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

Results:

(1, 'John', 'Smith', 50, datetime.date(1969, 4, 5), 1)

(2, 'Brian', 'Morgan', 38, datetime.date(1981, 2, 11), 1)

(3, 'David', 'White', 77, datetime.date(1942, 6, 30), 0)

Update Data

Running the code below will update the “active” column for the “David White” record from 0 (inactive) to 1 (active).

```
import sqlalchemy as db

def main():
  # get engine object using pymysql driver for mysql
```

```
   engine = db.create_engine("mysql+pymysql://username:password
@localhost/movie")
   meta_data = db.MetaData()
   # get connection object
   connection = engine.connect()
  # get actor table definition
   actor_table = db.Table("actor", meta_data, autoload=True,
autoload_with=engine)
  # set update sql statement. update column 'active' to true
where id is equal to 4 sql_query = ub.update(actor_table).
 values(active=True)
.where(actor_table.columns.id==4)
   results = connection.execute(sql_query)

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

Before the update, the result is as follows:

Result:

```
(3, 'David', 'White', 77, datetime.date(1942, 6, 30), 0)
```

After the update, the result is as follows:

Result:

```
(3, 'David', 'White', 77, datetime.date(1942, 6, 30), 1)
```

Delete Data

The code example will remove the “John Smith” record from the “actor” table.

```
import sqlalchemy as db

def main():
  # get engine object using pymysql driver for mysql
   engine = db.create_engine("mysql+pymysql://root:ana123!QAZ@localhost/
   movie")
   meta_data = db.MetaData()
  # get connection object
   connection = engine.connect()
   # get actor table definition
```

```
    actor_table = db.Table("actor", meta_data, autoload=True,
    autoload_with=engine)
    # set delete sql statement. delete the record where id is equal to 1
    sql_query = db.delete(actor_table).where(actor_table.columns.id==1)
    results = connection.execute(sql_query)

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

Before the deletion, the result is as follows:

Result:

(1, 'John', 'Smith', 50, datetime.date(1969, 4, 5), 1)

(2, 'Brian', 'Morgan', 38, datetime.date(1981, 2, 11), 1)

(3, 'David', 'White', 77, datetime.date(1942, 6, 30), 1)

After the deletion, the result is as follows:

Result:

(2, 'Brian', 'Morgan', 38, datetime.date(1981, 2, 11), 1)

(3, 'David', 'White', 77, datetime.date(1942, 6, 30), 0)

Summary

This unit explains and defines the most important Python libraries used today. Specific attention is dedicated to data ecosystem libraries developed for data analytics and machine learning/artificial intelligence projects. NumPy, pandas, matplotlib, SymPy, IPython, SciPy, Seaborn, scikit-learn, and Bokkeh libraries are covered in detail with plenty of sample code to demonstrate how to use them. The ways that Cython, PyPy, and Numba libraries can speed up Python programs are also explained. A Numba example code is provided to show how to speed up a NumPy two-dimensional array. Also covered was the Python SQLAlchemy SQL toolkit and how it can connect to a database engine and provide create, read, update, and delete operations.

Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!

Unit 5

Working With Python

STUDY GOALS

On completion of this unit, you will have learned...

- ... how to define a virtual environment and create one using the “venv” module.
- ... how to create a virtual environment with the Anaconda Distribution Package in Windows.
- ... how to manage Python packages using “pip” and “conda” modules.
- ... how to define unit and integration tests.
- ... how to use “unittest” testing framework.
- ... how to write comments in your source code using block, inline, and docstrings.

5. Working With Python

Introduction

One of the most important tools used by Python Application Developers today is virtual environment (VE). A VE is a simple tool that helps to keep the dependencies that are required by different projects separate by creating isolated Python virtual environments for them. The Python packages (libraries) are installed in that isolated environment instead of globally. One of the most popular programs used to create VE is the Anaconda Distribution Package which can be used on Windows, Unix, and MacOS.

The Python packages need to be installed based on the selected Python version. To manage these packages, Python uses "pip" and "conda" modules. These modules allow users to install, uninstall, and update any Python packages in the VE for the selected Python version.

Before deploying any computer programs to be used in production, they need to be tested so that any possible bugs can be removed. Python provides a "unittest" framework for unit and integration tests.

Documenting your source code is a must for any program today. Programming comments help application developers to understand the source code quickly for future maintenance and debugging purposes. Python uses the PEP 8 (Python Enhancement Proposal) as a model for the style of code and comments. Block, inline, and "docstring" comments are very useful for commenting Python code.

5.1 Virtual Environments

Python programming language has its own unique way of downloading, deploying, and storing packages (libraries). Every Python application has its own specific requirements. This means that it may not be possible for one Python installation to meet the requirements of every application. It is a good development practice to create **virtual environments** (VE) to manage separate versions of Python and their required libraries for specific project needs (Lutz, 2013).

Virtual environments
This refers to any software, program, or system that implements, manages, and controls multiple virtual environment instances.

A simple VE could be a self-contained directory tree that contains a Python installation for a particular version of Python, plus its required additional libraries. Python has many VE setup tools, and its most popular is called "venv." This module provides a good support for creating lightweight VEs with their own site directories, which are optionally independent from system site directories. Each VE has its own Python module files and can have its own separate set of installed Python packages in its site directories.

To create a VE, we run the following code from the command prompt:

```
python -m venv E:\my_env
```

The structure of the created VM "my_env" is shown below.

- Include folder — This includes modules if necessary.
- Lib folder — This contains a site-packages folder with Python libraries and packages based on the selected Python version.
- Scripts — This contains Python related exe files, activate and deactivate bat files, etc.
- pyvenv.cfg file — This is a Python VM configuration file that includes Python main install directory, version, etc.

```
home = C:\Users\ernest\Anaconda3
include-system-site-packages = false
version = 3.7.3
```

When the VE is created, it needs to be activated so that it can be used.

To activate the VE on Windows, run:

```
E:\my_env\Scripts\activate.bat
```

To activate the VE on Unix or MacOS, run:

```
source my_env/bin/activate
```

Activating the virtual environment will change your shell's command prompt to show which virtual environment you are using. For example, on Windows you will see

```
(my_env) C:\my_env\Scripts>
```

If you run a Python module with the following lines

```
import sys
print("python: {}".format(sys.version))
```

then the result will be

```
python: 3.6.9 |Anaconda, Inc.| (default, Jul 30 2019, 14:00:49) [MSC
v.1915 64 bit (AMD64)]
```

In this case, the VE was created with Python 3.6.9 version.

Setup a VE with Anaconda Distribution Package in Windows

One of the most popular Python VE management tools for Microsoft Windows is "conda," which is included in Anaconda Distribution packages. The 64-bit package distribution provides a simple way to set up a Python development environment. This

distribution tool includes the command prompt utility name “conda.” Conda is an open source command prompt package used to manage Python development environments on Windows, macOS, and Linux.

Follow the steps below to create a Python VE on Windows. First, make sure you have installed the 64-bit Anaconda Distribution package with Python 3.x version.

1. Open “Anaconda Prompt” from the Windows start button as “Administrator.”
2. Check “conda” version.
   (base) C:\WINDOWS\system32>conda –version
3. Updating conda is always recommended.
   (base) C:\WINDOWS\system32>conda update -n base -c defaults conda
4. Create a new environment named python3.6.9 with Python 3.6.9 release.
   (base) C:\WINDOWS\system32>conda create --name python3.6.9 python=3.6.9
   The information shown below will appear on the screen:

```
The following NEW packages will be INSTALLED:

    certifi:    2019.6.16-py36_1
    pip:        19.1.1-py36_0
    python:     3.6.9-h5500b2f_0
    setuptools:    41.0.1-py36_0
    sqlite:     3.29.0-he774522_0
    vc:      14.1-h0510ff6_4
    vs2015_runtime:   14.15.26706-h3a45250_4
    wheel:      0.33.4-py36_0
    wincertstore:  0.2-py36h7fe50ca_0

Proceed ([y]/n)?
```

5. Type “y” and press “Enter.” In your Anaconda envs folder you will see the python3.6.9 folder.
   (C:\your_anaconda_path\envs\python3.6.9)
6. Activate your python3.6.9 environment.
   (base) C:\WINDOWS\system32>activate python3.6.9
   The command prompt will change to “(python3.6.9) C:\WINDOWS\system32>”.
7. See Python installed packages.
   (python3.6.9) C:\WINDOWS\system32>conda list
8. See the list of all installed environments.
   (python3.6.9) C:\WINDOWS\system32>conda info --envs
9. Install the seven most popular Python packages:
   - NumPy
     (python3.6.9) C:\WINDOWS\system32>conda install numpy
   - Pandas
     (python3.6.9) C:\WINDOWS\system32>conda install pandas
   - SciPy
     (python3.6.9) C:\WINDOWS\system32>conda install scipy
   - StatsModels
     (python3.6.9) C:\WINDOWS\system32>conda install statsmodels

- Scikit-Learn
  (python3.6.9) C:\WINDOWS\system32>conda install scikit-learn
- Matplotlib
  (python3.6.9) C:\WINDOWS\system32>conda install matplotlib
- Seaborn
  (python3.6.9) C:\WINDOWS\system32>conda install seaborn

The most popular free-windows Python integrated development environments (IDE) are

- Spyder (it comes with Anaconda Distribution package),
- PyCharm, and
- Microsoft Visual Studio Code (it is also called VS Code).

Older, more traditional IDEs such as Eclipse and NetBeans are still popular today. For teaching and presentation purposes, the Python Jupyter Notebook web-based IDE is an excellent choice. The Jupyter Notebook project was developed to create a consistent set of open-source tools for scientific research, reproducible workflows, computational narratives, and data analytics. Those tools translated well to industry and today, Jupyter Notebooks have become an essential part of a data scientist's toolkit.

Write the following "versions.py" module and run it.

```
import sys
import pandas as pd
import numpy as np
import scipy
import sklearn
import matplotlib
import seaborn
import statsmodels

def main():
   print("python: {}".format(sys.version))
   print("numpy: {}".format(np.__version__))
   print("pandas: {}".format(pd.__version__))
   print("scipy: {}".format(scipy.__version__))
   print("scikit-learn: {}".format(sklearn.__version__))
   print("matplotlib: {}".format(matplotlib.__version__))
   print("seaborn: {}".format(seaborn.__version__))
   print("statsmodels: {}".format(statsmodels.__version__))

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

The following results were obtained on August 16th 2019. Some versions may be updated in the future.

```
python: 3.6.7 |Anaconda, Inc.| (default, Dec 10 2018, 20:35:02) [MSC
v.1915 64 bit (AMD64)]
numpy: 1.16.4
pandas: 0.23.4
scipy: 1.1.0
statsmodels: 0.9.0
scikit-learn: 0.20.1
matplotlib: 3.0.2
seaborn: 0.9.0
```

5.2 Managing Packages

To manage packages in Python, the following two libraries (packages) are used:

- pip — This is the preferred Python installer program used to install packages from the Python Package Index (PyPI). This package is a public repository of software for the Python programming language. It helps you to find and install software that has been developed and shared by the Python community.
- conda — This is the main package that can be used with command line commands at the Anaconda Prompt for Windows or in a terminal window for macOS or Linux (McKinney, 2017).

During a Python VE setup, it is recommended to use "conda" first. If conda cannot find the required package to be installed, use "pip" as second alternative. The commands of both packages are smart enough to check dependency on the correct package version for the selected Python version installed.

Managing Packages Using "pip"

In general, when you create your own Python VE, the "pip" library is already installed in the `your_python_path\Lib\site-packages` folder by default. We will now look at some "pip" main command lines.

Check "pip" version.

```
pip –version
```

Result:

```
pip 19.1.1 from C:\Snaps\AppData\Local\Continuum\anaconda3\lib\site-packages
\pip (python 3.7)
```

You can learn about pip supported commands by running it with the help command.

```
pip help
```

Result:

```
Usage:
 pip <command> [options]

Commands:
 install                     Install packages.
 download                    Download packages.
 uninstall                   Uninstall packages.
 freeze                      Output installed packages in requirements format.
 list                        List installed packages.
 show                        Show information about installed packages.
 check                       Verify installed packages have compatible dependencies.
 config                      Manage local and global configuration.
 search                      Search PyPI for packages.
 wheel                       Build wheels from your requirements.
 hash                        Compute hashes of package archives.
 completion                  A helper command used for command completion.
 help                        Show help for commands.

General Options:
 -h, --help                  Show help.
 --isolated                  Run pip in an isolated mode, ignoring environment
variables and
user configuration.
 -v, --verbose               Give more output. Option is additive, and can be used
up to 3 times.
 -V, --version               Show version and exit.
 -q, --quiet                 Give less output. Option is additive, and can be used
up to 3 times
(corresponding to WARNING, ERROR, and CRITICAL logging levels).
 --log <path>                Path to a verbose appending log.
 --proxy <proxy>             Specify a proxy in the form [user:passwd@]proxy.server:port.
 --retries <retries>         Maximum number of retries each connection should attempt
(default 5 times).
 --timeout <sec>             Set the socket timeout (default 15 seconds).
 --exists-action <action>    Default action when a path already exists:
(s)witch, (i)gnore, (w)ipe, (b)ackup, (a)bort.
 --trusted-host <hostname>   Mark this host as trusted, even though it does not have valid
or any HTTPS.
 --cert <path>               Path to alternate CA bundle.
 --client-cert <path>        Path to SSL client certificate, a single file containing the
private key and the certificate in PEM format.
 --cache-dir <dir>           Store the cache data in <dir>.
 --no-cache-dir              Disable the cache.
 --disable-pip-version-check
```

```
                              Don't periodically check PyPI to
determine whether a
new version of pip is available for download. Implied with --no-index.
 --no-color                 Suppress colored output
```

Upgrade pip.

```
python -m pip install --upgrade pip
```

Result:

Successfully installed pip-19.3.1

Install a package.

```
pip package_name install
```

See a list of packages installed in your VE.

```
pip list
```

Upgrade a package.

```
pip install --upgrade package_name
```

Remove (uninstall) a package.

```
pip uninstall package_name
```

Search for a package.

```
pip search package_name
```

Check status of a package.

```
pip show package_name
```

Managing Packages Using “conda”

In the previous section, the “conda” package was used to create a Python VE in Windows. We will now look at the most important “conda” commands that follow the same content used for the “pip” package.

Check “conda” version.

```
conda --version
```

Result:

conda 4.7.12

You can learn about conda supported commands by running it with the help command.

```
conda help
```

Result:

```
Options:

positional arguments:
 command
   clean       Remove unused packages and caches.
   config      Modify configuration values in .condarc. This is modeled
                after the git config command. Writes to the user .condarc
                file (C:\Users\ernest\.condarc) by default.
   create      Create a new conda environment from a list of specified
                packages.
   help        Displays a list of available conda commands and their help
                strings.
   info        Display information about current conda install.
   init        Initialize conda for shell interaction. [Experimental]
   install     Installs a list of packages into a specified conda
                environment.
   list        List linked packages in a conda environment.
   package     Low-level conda package utility. (EXPERIMENTAL)
   remove      Remove a list of packages from a specified conda environment.
   uninstall   Alias for conda remove.
   run         Run an executable in a conda environment. [Experimental]
   search      Search for packages and display associated information. The
                input is a MatchSpec, a query language for conda packages.
                See examples below.
   update      Updates conda packages to the latest compatible version.
   upgrade     Alias for conda update.

optional arguments:
 -h, --help    Show this help message and exit.
 -V, --version Show the conda version number and exit.

conda commands available from other packages:
 build
 convert
```

```
debug
develop
env
index
inspect
metapackage
render
server
skeleton
verify
```

Update conda.

```
conda list
```

Upgrade a package.

```
conda update package_name
```

Remove (uninstall) a package.

```
conda uninstall package_name
```

Search for a package.

```
conda search package_name
```

Comparison between “conda” and “pip” packages can be shown in the table below:

Comparison of Conda and Pip		
	conda	pip
Manages	binaries	wheel or source
Can require compilers	no	yes
Package types	any	Python-only
Create environment	yes, built-in	no, requires virtualenv or venv
Dependency checks	yes	yes

	conda	pip
Package sources	Anaconda repo and cloud	PyPI

5.3 Unit and Integration Testing

Software testing is part of the development life cycle for any computer program today. Software quality assurance (QA) engineers specialize in software testing (Beazley & Jones, 2013). The most common functional programs for testing used by software engineers today are as follows (Lutz, 2013):

- Unit tests — This test checks the functionality of the program and helps you to isolate what is broken in your program in order to fix it faster.
- Integration tests — This test checks many functionaries of the program and how they operate with each other. It also helps you to find as many broken functionalities as possible in your program by testing the program as a whole. In general, these tests require more time than the unit tests.

Below, we will look at some important rules of software testing.

- Start with a simple unit test first. This code is generally developed in function procedures located in a module or in a class file.
- Make sure each unit test is fully independent. Each test must be able to run alone and also within the complete test suite, regardless of the order that they are called.
- Try to make your unit test run quickly so that the development time will not increase.
- Run your function's tests frequently, ideally automatically when you save the code.
- Always run the full test suite before and after any coding session. This will guarantee that you did not break anything in the rest of the code.
- Before pushing code to a shared repository, make sure you run a session with all the case tests.
- If you need to interrupt your development session for any reason, it is good practice to write a broken unit test about what you want to develop next. It will help you to remember where you are when you come back to the program.
- If you find a bug in your program, it is a good idea to write a new test pinpointing this bug. This test will be part of your final program testing.
- Provide the correct names of your functions based on its implementation.
- End your program testing with many possible combinations of integrations tests. Make sure that they are planned and documented properly.

Unittest - Unit Testing Framework

The “unittest” framework is the most popular framework used in Python application development today (Matthes, 2019). Based on its documentation, it supports test automation, sharing of setup, shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. We will now examine the following unit test cases for math addition and subtraction.

Example:

```
import unittest

class MathOperations(object):

   def math_addition(self, number_1, number_2):
       '''
       provide math addition
       number_1: first number
       number_2: second number
       return: addition of first and second numbers
       '''
       result = number_1 + number_2
       return result

   def math_subtraction(self, number_1, number_2):
       '''
       provide math subtraction
       number_1: first number
       number_2: second number
       return: subtraction of first minus second numbers
       '''
       result = number_1 - number_2
       return result

class UnitTestMathOperations(unittest.TestCase):

   def test_addition(self):
       '''
       test math addition
       '''
       math_operations = MathOperations()
       result = math_operations.math_addition(2, 2)
       self.assertEqual(result, 4, "The addition should be 4")

   def test_subtraction(self):
       '''
       test math subtraction
       '''
```

```
        math_operations = MathOperations()
        result = math_operations.math_subtraction(2, 2)
        self.assertEqual(result, 0, "The subtraction should be 0")

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

Result:

```
Finding files... done.

Importing test modules ... done.



----------------------------------------------------------------------

Ran 2 tests in 0.000s



OK
```

There are two important things to note about the “UnitTestMathOperations” class test: First, this class is derived from the “TestCase” superclass. This “TestCase” class provides several assert methods to check for and report failures. Second, two “assertEqual()” assert methods were used to test for the numerical outcome of the mathematical operations. The following table shows the most commonly used methods in unit tests:

Commonly Used Methods in Unit Tests

Method	Checks that
`assertEqual(a, b)`	a == b
`assertNotEqual(a, b)`	a != b
`assertTrue(x)`	bool(x) is True
`assertFalse(x)`	bool(x) is False
`assertIs(a, b)`	a is b

Method	Checks that
`assertIsNot(a, b)`	a is not b
`assertIsNone(x)`	x is None
`assertIsNotNone(x)`	x is not None
`assertIn(a, b)`	a in b
`assertNotIn(a, b)`	a not in b
`assertIsInstance(a, b)`	isinstance(a, b)
`assertNotIsInstance(a, b)`	not isinstance(a, b)

If in the test function "`test_addition()`", the "`assertEqual()`" method is changed to "`self.assertEqual(result, 5, "The addition should be 4")`", the results of the test will be

```
Finding files... done.
Importing test modules ... done.

======================================================================
FAIL: test_addition (unit_test_example.UnitTestMathOperations)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\file_path\iubh_fernstudium\src\unit_test_example.py", line 28, in test_addition
    self.assertEqual(result, 5, "The addition should be 4")
AssertionError: 4 != 5 : The addition should be 4

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=1)
```

The test, of course, failed because 4 is not equal to 5.

5.4 Documenting Code

In computer program development, a programming comment is a programmer-readable explanation or annotation in the source code. They are added with the purpose of making the source code easier for developers to understand. These comments are ignored by compilers and interpreters of programming languages.

The main idea behind programming comments is to help application developers to understand the source code quickly for future maintenance and debugging purposes. Comments help other programmers to go through your code and gain an understanding of how it all works based on the program requirements. You can guarantee a smooth program transition by choosing to comment your code from the outset of a project. If a bug is found in a selected line of code, based on the comment, we can understand what happened and update the code.

Python programming language has a style guide that regulates the writing of code and comments. It uses the PEP (Python Enhancement Proposal) standards. The PEP is a design document that provides information to the Python community, describes a new feature of Python, or explains its processes or environment. The PEP should provide a concise technical specification and a rationale of the feature.

The PEP 8 -- Style Guide for Python Code document includes the main coding conventions for the Python code that make up the standard library in the main Python distribution (Ramalho, 2015). In this document, a “comment” topic is provided to specify its best practices. Below are some of them.

- Always prioritize keeping the comments up-to-date when the code has been updated.
- Comments should be complete, clear sentences, not just a collection of words.
- Block comments generally consist of one or more lines built from complete sentences.
- You should use two spaces after a sentence-ending period in multi-sentence comments, except after the final sentence.
- Write your comments in English, unless you are 100 percent sure that the code will be read by people who do not speak English.

Comments

Python provides three main types of comments in the code.

1. Block comments — These generally apply to all code that follows them, and are indented to the same level as that code. Generally, each line of a block comment starts with a # sign and a single space.

Example:

```
# this code will calculate the following financial taxes
# quarterly federal tax return 1
# quarterly federal tax return 2
# quarterly federal tax return 3
```

2. Inline comments — These are comments on top of, or on the same line as, a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.
   Examples:

```
# calculate quarterly federal tax by year and quarter. quarter
parameter is optional
result = quarterly_federal_tax(tax_year, tax_quarter_number=None)
result = quarterly_federal_tax(tax_year, tax_quarter_number=None)
# calculate quarterly federal tax by year and quarter. quarter
parameter is optional
```

3. Documentation strings — These comments are known as “docstrings” as defined in PEP 257. A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. The “docstrings” comments are not necessary for non-public methods, but they should have a comment that describes what they do in general. Generally, this comment should appear after the “def” line. It can use three single quotes ''' and three double quotes """.
   Example:

```
def math_addition(self, number_1, number_2):
       '''
       provide math addition
       number_1: first number
       number_2: second number
       return: addition of first and second numbers
       '''
       result = number_1 + number_2
       return result
```

Summary

Virtual environments (VE) are explained and defined in this unit. A VE is a self-contained directory tree that contains a Python installation for a particular version of Python, plus its required additional libraries. A setup of a VE with the Anaconda Distribution Package in Windows is covered in detail. The “conda” open source commands prompt package is used to manage Python development environments on Windows, macOS, and Linux. To demonstrate how to manage packages in Python, the “pip” and “conda” libraries were presented. The “unittest” framework is defined as the most popular framework used in Python application development today and is accompanied by a generic Python example code.

In computer program development, programming comments are a programmer-readable explanation or annotation in the source code. This is known as a program development documentation. It is very important to follow Python PEP 8 (Python Enhancement Proposal) for the style regulations of code and comments.

Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!

Unit 6

Version Control

STUDY GOALS

On completion of this unit, you will have learned...

- ... what version control is and why it is important.
- ... about the three types of VC systems.
- ... about the most useful Git VC systems concepts.
- ... how to use Git VC main command operations.
- ... about the most popular Git hosting services available today.

6. Version Control

Introduction

Creating code is a dynamic process that involves exploring and trying out different approaches in order to reach the intended goal. Thus, having a system or tool that aids in tracking different code variants and enabling the developer to revert back to previous versions is extremely helpful. This becomes even more evident when a whole developer team works together on a piece of software where each team member makes changes to the code base independently of the others. Here, the situation often occurs that different developers make changes to the same part of a program without knowing about each other. Typcially, these changes are consistent with the rest of the work of the respective developer, yet in conflict with the changes made by the other.

Version control software is a tool that addresses these challenges in large scale software development. It works by keeping a record of all changes to a code base in a specialized type of database. This tracking mechanism makes reverting back to a functional state of the program easier after an error has been found. Moreover, it helps to highlight code conflicts and enables a resolution that is compatible with all other pending changes.

In this unit, version control systems will be defined and the three main types (local, centralized, and decentralized systems) will be presented. Concurrent versions system (CVS), Apache Subversion (SVN), and Git will also be explained.

As Git is the most popular VC used system today, this unit will cover the main VC system concepts. These are Git Linux and Windows installation, Git general team member workflow, main command operations, and popular Git hosting services available in the market today.

6.1 Introduction to Version Control

Version control
This is the management of changes to documents, computer programs, large web sites, and other collections of information.

Version control (VC) is an important component of software configuration management. Sometimes VC is referred to as revision control or source control. In general, source control is specific to tracking changes in source code. Changes are usually identified by a number or letter code, termed the “revision number,” “revision level,” or simply “revision.” For example, an initial set of files is “revision 1.” When the first change is made, the resulting set is “revision 2,” and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged. VC has more general purposes including source code, digital assets, etc.

VC is very important in the business application development lifecycle. It maintains a single source code of truth for development teams, helps to facilitate collaboration, and accelerates program release velocity. It allows multiple application developers on the team to work on the same source code base. They can add, update, or merge any

source code at any given time. The team members may be geographically dispersed and may pursue different and even contrary interests, meaning that sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be indispensable in such situations.

It is important to mention that the VC may also track the changes within application configuration files, database objects, developer and user documentations, etc. This function is generally used by system and database administrators.

Since the code base is the same, it also requires specific (read/write/execute) permission to be granted to a set of developer team members. A lead developer is then in charge of managing permissions so that the code base is not compromised.

Manually setting up a VC environment is not an easy task. Many software systems have been developed to automate the VC process. There are three types of VC developed systems.

- Local version control systems — These are based on copying files locally in a time-stamped folder in your computer. With this approach, it is easy to forget which folder you are in and you may accidentally write to the wrong file or copy over files you do not mean to. This system provides no possibility for collaboration at all. The main issue is that if your local hard drive becomes corrupted or fails, the whole project source code history will be destroyed.
- Centralized version control systems — These are based on a single server that contains all the versioned files, and a number of clients that check out files from that network central place. For many years, this has been the standard for VC. In these systems, every team member knows, to a certain degree, what everyone else on the project is doing. System administrators have good control over who can do what. It is easier to manage a central VC server than local databases on every client. As with local VS systems, if the central hard drive crashed for some reason, everything from the project would be gone.
- Distributed version control systems — These are based on many clone servers. In this case, the team members mirror the complete source code data structure (repository), including the full history. If any server crashes and these systems were collaborating via that server, any of the team member repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the source code data structure. This is the most popular approach for developing VC systems today (Mohabia, 2018).

Some of the most common open source VC systems used today are Git, CVS, SVN, etc.

- CVS (concurrent versions system) — This is a traditional, open source, centralized VC system. CVS automates the storing, retrieval, logging, identification, and merging of revisions. It is very useful for text that is revised frequently, including source code, programs, documentation, papers, and form letters. It uses a client-server architecture (distributed application structure that partitions tasks or workloads between the providers of a resource or service called servers, and service requesters called

clients) where the server stores the current version(s) of a project and its history, and the client can connect to the server to check-out a complete copy of the project and check-in project changes.
- SVN (Apache subversion) — This is a software versioning and revision control system distributed as open source under the Apache license. Software developers use subversion to maintain current and historical versions of files such as source code, web pages, and documentation. Subversion was designed to operate across networks, which allows it to be used by developer team members on different computers. It fosters collaboration and allows various people to modify and manage the same set of data from their respective locations. Because of this, source code updates can occur more quickly without a single conduit through which all modifications must occur.

6.2 Version Control With Git

Git
This is a distributed version-control system for tracking changes in source code during software development.

Git is the most popular distributed VC system today. In Git, source control changes do not have to be committed to the same central repository, which would require every team member working on the project to access that central repository and download the latest code in order to save changes. Instead, every team member can have their own localized repository with its entire history (Matthes, 2019). It was developed to meet the following goals:

- a simple design for easy maintenance,
- strong support for non-linear development with many parallel independent lines of source code, and
- a completely distributed system.

We will now look at the main VC system concepts (Rother, 2017).

- Repository is a central directory location used to store multiple versions of files. It is a simple hidden folder named ".git" in the root directory of your project. Repositories can be local and remote.
- Local repository is a local ".git" directory inside your project's root directory in your computer. A team member is the only person that can work with this repository by committing changes to it. It could be any selected directory in your computer.
- Remote repository is a remote ".git" folder that is typically located on a remote server on the internet or in your local network. There are no actual working files associated with a remote repository as it has no working directory but it exclusively consists of the ".git" repository folder. Team members are using remote repositories to share and exchange data. They serve as a common base where everybody can publish their own changes and receive changes from their team members.
- Blobs (binary large objects) are versions of a file. A blob holds the file data but does not contain any metadata about the file. It is a simple binary file in the Git database.
- Trees are objects that represent a directory. They hold blobs as well as other subdirectories. A tree is a binary file that stores references to blobs and trees.

- Commits hold one state of the repository. A commit object is a node of the linked list. Every commit object has a pointer to the parent commit object. From a given commit, you can traverse back by looking at the parent pointer to view the history of the commit. If a commit has multiple parent commits, then that particular commit has been created by merging two branches.
- A branch is an independent line of development taken from the same source code. Different branches can be merged into any one branch as long as they belong to the same repository. By default, Git has a master branch. Generally, a branch is created to work on a new program feature. Once the feature is completed, it is merged back with the master branch and it is deleted.
- Tags are meaningful names with specific versions in the repository. Tags are very similar to branches, but the difference is that tags are immutable. Once a tag is created for a particular commit, even if you create a new commit, it will not be updated. Usually, tags are created for product releases.
- Clone is an operation used to create the instance of the repository. Clone operation checks out the working copy and mirrors the complete repository. Team members can perform many operations with this local repository. The only time networking gets involved is when the repository instances are being synchronized.
- Pull is an operation used to copy the changes from a remote repository instance to a local one. The pull operation is used for synchronization between two repository instances. This operation is used very frequently.
- Push is an operation used to copy changes from a local repository instance to a remote one. This is used to store the changes permanently in the Git repository. This operation is used very frequently.
- HEAD is a pointer to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit. This is a unique code.
- Merge is an operation that commits one location to another location. Whether branches are created for testing, bug fixes, or other reasons, merge can commit changes to other branches. Merging takes the contents of a source branch and integrates them with a target branch. In this process, only the target branch is changed. The source branch history remains the same.
- Rebase is another way to integrate changes from one branch to another. Rebase compresses all the changes into a single “patch” and then integrates the patch with the target branch. Unlike merging, rebasing flattens the history because it transfers the completed work from one branch to another. In the process, unwanted history is eliminated.
- Diff is an operation used to compare different versions of your files. It shows the changes made to any files with different versions. The most common scenario in which to use diff is to see the changes that you made after your last commit.
- Revision is a small integer. It represents the version of the source code made by the commit command. It is a strictly local convenience identifier for a revision. It can be useful as it is shorter to type than the 40-digit hexadecimal string that uniquely identifies a revision.
- URL represents the weblink location of the Git repository. Git URL is stored in a configuration file. It must be unique for each Git repository (Chowdhury, 2019).

Git

To install Git for Linux computers (Debian base system), use the following code (Williamson & Olsson, 2014):

```
$ sudo apt-get install git-core
```

For Windows, download the file shown below and run it.

```
Git-2.24.0.2-64-bit.exe
```

Three-basic workflow of Git
We will now look at the three-basic workflow of Git, including working directory, staging area and git repository.

- Step 1: Files are modified by the working directory.
- Step 2: Add these files to the staging area.
- Step 3: Perform a commit operation that moves the files from the staging area. After the push operation, it stores the changes permanently to the Git repository.

Git general team member workflow
In general, Git workflow for a team member contains the following simple steps:

- Clone the required remote Git repository as a local working copy in your machine.
- Modify the local working copy by adding/editing any necessary files.
- If necessary, the local working copy can be updated by taking other developers' changes.
- Review the changes before the commit.
- Commit your changes. If everything is fine, then you push the changes to the remote repository.
- After committing, if something went wrong, the last commit must be corrected and then you should push the changes back to the remote repository again.

Git main command operations

1. Create a new repository — Open Git and run the command
   `git init`

2. Checkout a repository — Create a working copy of a local repository by running the command
   `git clone /path/to/repository`

   When using a remote server, your command will be
   `git clone username@host:/path/to/repository`

3. Add and commit — Changes can be proposed by adding it to the index using the following code:

```
git add <filename>
git add *
```

This is the first step in the basic git workflow. To actually commit these changes, use the code below.

```
git commit -m "Commit message"
```

Now the file is committed to the HEAD, but it is not in your remote repository yet.

4. Pushing changes — The changes are now in the HEAD of your local working copy. To send those changes to your remote repository, execute

   ```
   git push origin master
   ```

   Change master to whatever branch you want to push your changes to.
   If you have not cloned an existing repository and want to connect your repository to a remote server, you need to add it with

   ```
   git remote add origin <server>
   ```

   Now you are able to push your changes to the selected remote server.

5. Create a new branch name and switch to it by using

   ```
   git checkout -b branch_name
   ```

   switch back to master

   ```
   git checkout master
   ```

   and delete the branch again.

   ```
   git branch -d branch_name
   ```

   A branch is not available to others unless you push the branch to your remote repository

   ```
   git push origin <branch>
   ```

6. Update and merge — Update your local repository to the newest commit, execute

   ```
   git pull
   ```

   To fetch and merge remote changes in your working directory, merge another branch into your active branch (e.g. master), using

   ```
   git merge <branch>
   ```

   In both cases, Git tries to auto-merge changes. Unfortunately, this is not always possible and results in conflicts. You are responsible for merging those conflicts manually by editing the files shown by Git. After changing, you need to mark them as merged with

   ```
   git add <filename>
   ```

   Before merging changes, you can also preview them by using

   ```
   git diff <source_branch> <target_branch>
   ```

7. Tagging — Use for software releases. You can create a new tag named 1.2.0 by executing
   ```
   git tag 1.2.0 1b1e3d655ff
   ```

   The 1b1e3d655ff stands for the first ten characters of the commit id you want to reference with your tag.
8. Log — Use to study repository history
   ```
   git log
   ```

   You can add many parameters to make the log look however you like. To only see the commits of a certain author, use
   ```
   git log --author=bob
   ```

   To see a very compressed log where each commit is one line, use
   ```
   git log --pretty=oneline
   ```

   To see an ASCII art tree of all the branches, decorated with the names of tags and branches, use
   ```
   git log --graph --oneline --decorate –all
   ```

   To see which files have changed, use the following
   ```
   git log --name-status
   ```

   These are just a few of the possible parameters you can use.
   ```
   git log --help
   ```

9. Replace local changes — To replace local changes, use the command
   ```
   git checkout -- <filename>
   ```

   This replaces the changes in your working tree with the last content in HEAD. Changes already added to the index, as well as new files, will be kept. If you want to cancel all of your local changes and commits, fetch the latest history from the server and point your local master branch.
   ```
   git fetch origin
   git reset --hard origin/master
   ```

   (Spini & Marino, 2015).

Popular Git hosting services

There are several Git repository hosting services, but not all of them provide a free option in their package. Below, we will look at the three most popular today.

- Github “is the largest community website for software development. It supports some of the best tools for issue tracking, code review, continuous integration, and general code management. And its underpinnings are still on Git, everyone's favorite

open source distributed version control system" (van Gumster, 2018, para. 3). GitHub provides free public and paid private repositories. It was acquired by Microsoft in 2018.
- GitLab is a fully open source hosting service. "You can host your code right on GitLab's site, much like you would on GitHub, but you can also choose to self-host a GitLab instance on your own server and have full control over who has access and how things are managed" (van Gumster, 2018, para. 4). GitLab has almost the same features as GitHub.
- Bitbucket is a good reflection of GitHub and GitLab. It "shares most of the features available on GitHub and GitLab, plus a few novel features of its own, such as native support for mercurial repositories" (van Gumster, 2018, para. 5).

Of the three hosting services, Github currently has the largest user base. The platform adds numerous tools for issue tracking, code review, continuous integration, and general code management to the base version management features provided by Git. There are free and paid tiers for project hosting on offer. A similar feature set is offered by GitLab. What sets GitLab apart is its support for self-hosting instances of the service. Thus, code does not have to be placed on external infrastructure. Bitbucket is a Git hosting service that is provided by Atlassian (n.d.) — the well-known maker of issue tracking and documentation tools Jira and Confluence. The feature set on offer is comparable to those of the previous services although it sets itself apart with the ability to use the Mercurial version control system as an alternative to Git (van Gumster, 2018).

Summary

Version control (VC) systems are important software development tools that are included in the software development lifecycle. This unit defines version control and why it is important to use it. The three types of VC systems (local, centralized, and distributed) are presented. Git is the most popular distributed VC system today. In addition, this unit covers the main VC system concepts including repository (local and remote) blobs, trees, etc. Installing Git and basics Git commands operations are also presented in this unit.

Knowledge Check

Did you understand this unit?

You can check your understanding by completing the questions for this unit on the learning platform.

Good luck!

Congratulations!

You have now completed the course. After you have completed the knowledge tests on the learning platform, please carry out the evaluation for this course. You will then be eligible to complete your final assessment. Good luck!

Appendix 1

List of References

List of References

Atlassian. (n.d.). *What is version control?* https://www.atlassian.com/git/tutorials/what-is-version-control

Beazley, D., & Jones, B. K. (2013). *Python cookbook* (3rd ed.). O'Reilly Media.

Chowdhury, K. (2019, December 10). *Git tutorial for beginners (Git bash commands)*. https://www.kunal-chowdhury.com/p/git-cheatsheet.html

Copeland, R. (2008). *Essential SLQAlchemy*. O'Reilly Media.

Kapil, S. (2019). *Clean python: Elegant coding in python*. Apress.

Lutz, M. (2013). *Learning python*. O'Reilly Media.

Matthes, E. (2019). *Python crash course: A hands-on, project-based introduction to python* (2nd ed.). No Starch Press.

McKinney, W. (2017). *Python for data analysis* (2nd ed.). O'Reilly Media.

Mohabia, S. (2018, May 28). *Up and running with version control systems*. Medium. https://medium.com/@shashankmohabia/up-and-running-with-version-control-systems-88c9857abe09

Ramalho, L. (2015). *Fluent python: Clear, concise, and effective programming*. O'Reilly Media.

Rother, K. (2017). *Pro python best practices: Debugging, testing and maintenance*. Apress.

Spini, F., & Marino, E. (2015, March 5). *Git—the simple guide*. Github. https://github.com/webcrumbs/git-crumbs/blob/master/git/Readme.md

van Gumster, J. (2018, August 30). *6 places to host your git repository*. Opensource.com. https://opensource.com/article/18/8/github-alternatives

Williamson, T., & Olsson, R. A. (2014, February). PySy: A python package for enhanced concurrent programming. *Concurrency and Computation—Practice & Experience, 26*(2), 309—335. https://doi.org/10.1002/cpe.2981.

Appendix 2

List of Tables and Figures

List of Tables and Figures

List of Tables and Figures

IUBH Internationale Hochschule GmbH
IUBH International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing address:
Albert-Proeller-Straße 15-19
D-86675 Buchdorf

Phone: +49 30 311 988 55
media@iubh.de