LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 1 INTRODUCTION

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

1. INTRODUCTION TO SOFTWARE DEVELOPMENT IN PYTHON

BY FRED ANNEXSTEIN

In this chapter we will cover the basic tools and terminology used in Python programming. We will first examine installing a Python Programming platform, which is used throughout this book. We then turn to look at the Python interpreter and the dynamic typing that underlies the Python execution model.

CHAPTER 1 OBJECTIVES

By the end of this chapter, you will be able to...

- Install and run the Anaconda version of the Python Interpreter.
- Recognize and describe the Python Interpreter employing common Python terminology.
- Practice running small Python scripts. Understand the Python software development process.
- Understand the Python runtime model including, dynamic typing, statements and expressions, and basic control.

- Understand the syntax and semantics of Python doctests. Formulate and execute a Python program with a variety of doctests.

THE ANACONDA PYTHON DISTRIBUTION

In this book we use the Anaconda Python distribution because it is easy to install on the most platforms such as Windows, macOS and Linux. Anaconda supports the latest versions of Python, the IPython interpreter, and Jupyter Notebooks. Anaconda also includes other software packages, libraries, and tools commonly used in Python programming and data science, allowing students to focus on learning the Python language and computing skills. In this book we will use the Spyder application, which is packaged with Anaconda, as the default software development environment. This environment will allow you to easily experiment and test for your Python programming exercises.

Download the Anaconda3 installer for either Windows, macOS or Linux from:
`https://www.anaconda.com/download/`
When the download completes, run the installer and follow the on-screen

instructions. To ensure that Anaconda runs correctly, do not move any of the files after you install it.

UPDATING ANACONDA

The next task is to ensure that Anaconda is up to date. Open a command-line window on your system as follows:

On macOS or Linux, open a Terminal from the Applications folder's Utilities subfolder.

On Windows, open the Anaconda Prompt from the start menu. Execute the Anaconda Prompt as an administrator by right-clicking, then selecting More > Run as administrator. If you cannot find the Anaconda Prompt in the start menu, simply search for it at the bottom of your screen.

In your system's command-line window, execute the following two commands to update Anaconda's installed packages to their latest versions:

```
conda update conda
conda update --all
```

INSTALLING THE OTHER PACKAGES

Anaconda comes with approximately 300 popular packages, such as NumPy, Matplotlib, pandas, Regex, SciKit-Learn, Seaborn, and many more. The number of additional packages you'll need to install throughout the book will be small and we'll provide installation instructions as necessary. As you discover new packages, their documentation will explain how to install them. Generally, you should not use the pip program, rather rely on the %conda install option to install additional packages.

A CALCULATOR

Let us now use the IPython interactive mode to evaluate simple arithmetic expressions. First, open a command-line window on your system: In the command-line window, type ipython, then press Enter (or Return). You should see text like the following, but this varies by platform and by IPython version:

```
fred$ ipython
Python 3.8.13 (default, Mar 28 2022, 06:16:26)
```

```
Type 'copyright', 'credits' or 'license' for
more information IPython 8.3.0 -- An enhanced
Interactive Python. Type '?' for help.
In [1]:
```

The text "In [1]:" is a prompt, indicating that IPython is waiting for your input. So now enter text to evaluate expressions:

```
In [1]:  25 * 4
Out[1]: 100

In [2]: 25 / 4
Out[2]: 6.25

In [3]: 25 // 4
Out[3]: 6

In [4]: 25 %  4
Out[4]: 1

In [5]: 25 **  4
Out[4]:390625
```

Python uses the asterisk (*) for multiplication, forward slash (/) for division, double slash (//) for integer division, percent (%) for modular arithmetic, and double star (**) for exponentiation. Parentheses should be used to force the evaluation order. Numbers with decimal points, like 12.7, 43.5 and 21.75, are called floating-point numbers.

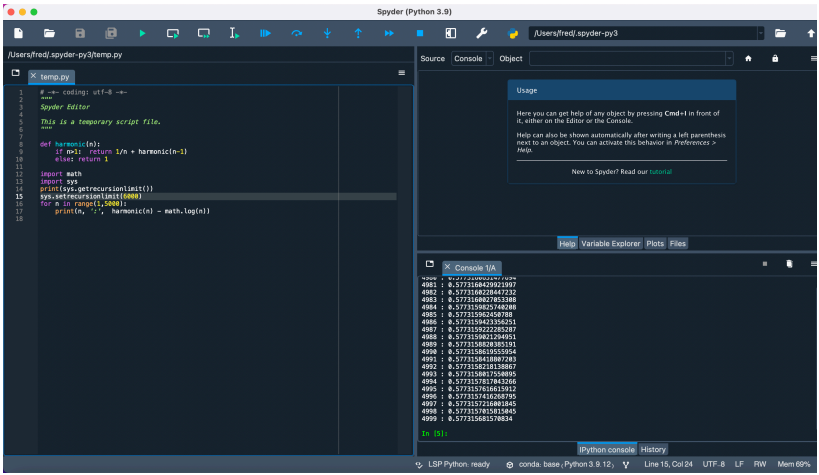
EXITING INTERACTIVE MODE

To leave interactive mode, you can:

- Type the exit command at the current prompt and press Enter to exit immediately.
- Type the key sequence <Ctrl> + d (or <control> + d) . This displays the prompt "Do you really want to exit ([y]/n)?". The square brackets around [y] indicate the default response—pressing Enter submits the default response and exits.
- Type <Ctrl> + d (or <control> + d) twice (macOS and Linux only).

CREATING SCRIPTS

Typically, you create your Python source code in an editor that enables you to type



text. We will use the Spyder application (which comes with Anaconda), which you can run from the terminal as follows:

```
fred$ spyder
```

The interactive terminal is called a repl for “read eval print loop” - and found in the lower right pane. The editor on the left side pane is used to type and save programs. Spyder provides many tools that support the entire software-development process, such as debuggers. In the upper right pane, Spyder has a set of linting tools for code analysis that detects style issues indicating potential bugs, and can give your code an overall quality score.

THE PYTHON INTERPRETER

Python code is usually saved as a text file with a name like 'lab1.py'. To run the code there is application program installed on your named "ipython". Other installations may call this "python3" or simply "python", and its job is reading and running your Python code line by line. This type of program is called an "interpreter". We will run the interpreter

directly within the Spyder application found in lower right pane.

TALKING TO THE INTERPRETER

We can use the interpreter to look up various information about the programs we are developing, including the formal descriptions built-in python functions. We find that just typing an experiment in the interpreter to see what it does is quick and satisfying way to get the information you need. You should replicate and then play with the examples in the book. Experiments will help you learn the material.

For simplicity we will denote a generic python interpreter prompt with >>>. This prompt will also be used to construct doctests. The symbol means the interpreter is waiting for you to type a line of Python code.

INDENTATION FOR THE INTERPRETER

It is possible to write multi-line indented code in the interpreter. If a line is not finished, such as ending with ":", typing the

return key does not run the code right away. You can write further indented lines. When you enter a blank line, the interpreter runs the whole thing. The python standard is to indent 4 spaces. This indentation should be done automatically when you type return. Hitting return on a blank line ends the block and runs the script. Try this example in your interpreter:

```
>>>  for i in range(5):
   ...:    mult = i * 100
   ...:    print(i, mult)
   ...:
0 0
1 100
2 200
3 300
4 400
```

HELP AT THE COMMAND PROMPT

When you type a question mark ? at the prompt you will get some basic help information. What about Python built-in data types? The built-in string data type known as <str> has many associated function or methods. We can pull down a list of these

function directly from the interpreter as follows.

```
>>> dir(str)
['__add__',
 '__class__',
 '__contains__',,....
```

With the built-in help function, you can pull down the docs for each function. To refer to a function associated with the str type, we use the form str.find. Note that the function name is not followed by parenthesis.

```
>>> help(str.find)
Help on method_descriptor:

find(...)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring
sub is found,such that sub is contained within
S[start:end].  Optional arguments start and end
are interpreted as in slice notation. Return -1
on failure.
```

USAGE OF NAMES

Variables and simply *names* in Python refer to values which are python objects. Names are like pointers in that they don't refer to specific locations in memory. Assignment

statements are used to make names refer to particular values. Many names can refer to one value or object, and in such a case are called *aliases*. Assignment statements never copy data, since they only reassign names to new values. Users of Python should be consciously aware when a name refers to *mutable or immutable data*. When a name refers to a value that is mutable -- the value can be changed without changing the name which is an object reference. Therefore all names that refer to that object will reflect the change to the value -- updates are therefore visible via all aliased names. When an object is immutable, no visible change is possible. Errors are raised when attempts at modifying an immutable value are executed.

PYTHON PACKAGE TERMINOLOGY

Software is complex and spans many layers of abstraction. Python simplifies these abstraction layers. When speaking about Python programming we say: Applications are collections of programs --- Programs are collections of modules --- Modules are collections of statements --- Statements are collections of expressions --- Expressions

create new objects or process existing objects.

Modules are the basic unit of code reusability in Python, and represents a block of code that may be imported by some other program. Three types of modules concern us here: pure Python modules, extension modules, and packages.

A pure module is a program that is a single .py file. An extention module may be written in the low-level language (e.g. C) of the Python implementation. A package is a special module that contains other modules; typically contained in a directory in the filesystem and distinguished from other directories by the presence of a file named __init__.py.

IMPORT AND FROM

Generally, when you want to add a package you will prefer to use import. Python programs are composed of multiple module files linked together by import statements, and each module has attributes -- a collection of variable names—usually called a namespace.

Importing everything can be convenient, especially when using interactively, but be careful when mixing with other libraries. To avoid inadvertently overriding other functions or objects, explicitly import only the needed objects. For example, take the module mpmath for high-precision decimal numbers. We can import the entire module, and access the sin and cos methods, as follows:

```
>>> import mpmath
>>> y = mpmath.sin(1)
>>> x = mpmath.cos(1)
```

Mpmath uses a global working precision; it does not keep track of the precision or accuracy of individual numbers. Performing an arithmetic operation or
calling mpf() rounds the result to the current working precision. The working precision is controlled by a context object called mp, which we can view as follows:

```
>>> from mpmath import mp
>>> print(mp)
```

We can also set the number of digits of precision accuracy, and produce a number

requiring many digits, before repeating, as follows:

```
>>> mp.dps = 3000
>>> mp.mpf(1) / mp.mpf(999**2)
```

Setting the precision to 3000 digits and should produce an interesting result.

PYTHON WORKFLOWS: DEBUGGING AND TESTING

Much of the effort in programming is focused on testing and debugging code, which involves learning to interpret errors and diagnose the cause of unexpected errors. Dividing by zero is not allowed and results in the raising of an exception—which is a sign that a problem occurred:

```
>>> 1/(1-1)
Traceback (most recent call last):
  Input In [] in <cell line: 1>
    1/(1-1)
ZeroDivisionError: division by zero
```

In the above example the execution of the interpreter is said to *raise an exception* of type ZeroDivisionError. When an exception is raised in IPython it will terminate the snippet,

then display the exception's traceback, and finally shows the next prompt so that you can input the next snippet.

Learning and working with the exception tracebacks that Python generates is good practice for debugging. Here are some more guiding principles.

Test incrementally: Every well-written program is composed of small, modular components that can be tested individually. Try to test everything you write as soon as possible to identify problems early and gain confidence in your components.

Isolate errors: An error in the output of a statement can typically be attributed to a particular modular component. When trying to diagnose a problem, trace the error to the smallest fragment of code you can before trying to correct it.

Check your assumptions: Interpreters do carry out your instructions to the letter — no more and no less. Their output is unexpected when the behavior of some code does not match what the programmer believes (or assumes) that behavior to be. Know your assumptions, then focus your debugging

effort on verifying that your assumptions actually hold.

Consult others: You are not alone! If you don't understand an error message, ask a friend, instructor, or search engine. If you have isolated an error, but can't figure out how to correct it, ask someone else to take a look. A lot of valuable programming knowledge is shared in the process of group problem solving.

In this book we will use the process of test driven development (TDD), which is simply software development process that puts emphasis on clearly documented code tests. Your goal is to place readable tests right into your everyday coding practice. For this course we will use a python package called the *doctest module*. This module allows us to run testing comments just like we run code.

COMMENTS IN PYTHON

All comments begin with a hash mark (#) and continue to the end of the line. Because comments do not execute, when you run a program you will not see any indication of the comment there.

Comments are in the source code for humans to read, not for computers to execute. Python does not have a syntax for multi line comments, however your IDE should allow insertion and removal of block comments. Strings can also be used for comments. If we do not assign strings to any variable name, then they play a role and act as comments.

DOCSTRINGS IN PYTHON

Docstrings are simply strings that occur as the first statement in a module, function, class, or method definition. Any such string is called a docstring and becomes the __doc__ special attribute of that object.

There is a standard python module called “doctest” that is useful for setting up a TDD testing framework. The doctest module searches for pieces of text that look like interactive Python sessions inside of the documentation parts of a module, and then executes (or re-executes) the commands of those sessions to verify that they work exactly as shown, i.e. that the same results can be achieved. In other words: The comments or help text of the module is parsed and run through the python

interpreter sessions. These slices of example code are run and the results are compared against expected values indicated in the tests.

HOW TO USE DOCTESTS

To use the “doctest module” it has to be first imported. The part of an interactive sessions with the example test code and the associated expected outputs must be copied inside of the docstring of the corresponding function.

SAMPLE RUN OF DOCTEST

Here is a small function with several doctests. Each test checks whether the code for the factorial function produces expected results. Try to load this code and run it to see the output of the doctests. The last if statement is boilerplate code, and the typical way we will execute the doctests when running any code.

```
def factorial(n):
    """Return the factorial of n >= 0.
    Here are three doctests:
    >>> factorial(3)
    6
    >>> factorial(30)
    265252859812191058636308480000000
```

```
    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    """

    f = 1
    for i in range(2,n+1):
        f *= i
    return f

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

CHAPTER 1 EXERCISES

1. Copy the code with the three doctests from above, and paste and load it into the Spyder IDE (left pane). Run the code to show the results of passing doctests. Your output should be as follows:

```
Trying:
    factorial(3)
Expecting:
    6
ok
Trying:
    factorial(30)
Expecting:
    265252859812191058636308480000000
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
1 items had no tests:
    __main__
```

```
1 items passed all tests:
   3 tests in __main__.factorial
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
```

2. Download the expanded doctest example from the following website: https://repl.it/@FredAnnexstein/doctest-example#main.py.

Consider this code and see that it contains examples with logic to raise exceptions based on potential input values to the function. Exceptions may be raised by the system, as in the ZeroDivisionError example above, or the user can raise exceptions based on the value of an object. For example, consider:

```
if math.floor(n) != n:
        raise ValueError("n must be exact integer”)
```

The code snippet above is used to test if a value n is an integer or not. If it is not, then an exception is raised as a ValueError. We will in a subsequent chapter that when an exception is raised it can be handled in a way that does not terminate the processing.

Modify the factorial() function by adding a snippet of code, such as

```
if “n is very large”:
        raise ValueError("n must not be so large!”)
```

so that when n is very large a ValueError exception is raised. You should determine the logic of the “n is very large” trigger condition. Run the doctest to test your results.

3. (Homework #1) From Canvas download the the folder for the Lab 1 on Expressions and Control. Extract the files in the folder if needed and open the index.html file in you browser. The Lab sheet should be formatted with a Red Banner on top. Complete the worksheet by turning in a file called lab1.py with your answers to all the required questions.

LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 2: FUNCTIONAL PROGRAMMING

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

2. FUNCTIONAL PROGRAMMING

BY FRED ANNEXSTEIN

CHAPTER 2 OVERVIEW

In this chapter we will be covering an introduction to the practice of functional programming. Functional programming gives you capabilities to use abstractions and compositions. Abstractions allow you to handle complex data in an intuitive way, and compositions allow you to develop software in a top-down or bottom-up fashion to meet your needs. We cover the basics of defining or constructing functions and the role of higher-order functions in software development.

CHAPTER 2 OBJECTIVES

By the end of this chapter, you will be able to….

- Identify the role of functional programming in the software development process.
- Justify the concept of pure functions and function objects in Python.
- Understand the function call stack and scope rules used in Python.

- Understand the syntax of call expressions and lambda expressions
- Understand and apply the concept of higher-order functions.
- Formulate a dispatch-function employing higher-order functions.
- Gain experience with programming in a functional style.

FUNCTIONAL PROGRAMMING

Python allows for multiple approaches to programming. The functional-programming (FP) style of programming is an important discipline to master. FP significantly differs from Object-oriented programming (OOP) which we will cover in Chapter 6. FP prevents many errors from creeping into code by avoiding coding side-effects. Such side-effects often result in unexpected state changes.

As the computational tasks you perform get more complicated, your code can become harder to read, debug and

modify, and therefore more likely to contain errors. Specifying how you want the code to work can become complex. A functional-style of programming lets you simply say what you want to do. It hides many details of how to perform each task. Typically, library code handles the how for you. As you'll see, this can eliminate many errors.

Simply stated, FP style allows you to decompose a problem into a set of functions. Ideally, the functions you write will only take inputs and produce outputs, and don't have any internal state that may impact the rest of the computation. Python works well with this style of programming, however it is not strictly a purely functional programming language. Well-known purely functional languages include the ML, OCaml, and Haskell.

PURE FUNCTIONS VS. NON-PURE FUNCTIONS

With every function you write you need to pay attention to the domain, that is the values for which the function is defined. And

also the range, that is the values that are potentially returned by the function.

A pure function - ONLY produces a return value with no side effects and always evaluates to the same result, given the same argument value(s).

A non-pure function - may produce some side effects, such as printing to the screen, and may not always evaluate to the same result, given the same argument values.

There are theoretical and practical advantages to using only pure functions when practicing the functional style of programming. For example it is easier to construct a mathematical proof that a functional program is correct. Proofs of correctness are different from testing a program on numerous inputs and concluding that its output is usually correct. Functional programs typically are more modular, that is programs are composed of small functions are easier to read and to check for errors. Functional programs are often created by arranging existing functions

in a new configuration and writing a few functions specialized for the current task.

EXPRESSIONS

We begin with python expressions, which describe a computation and evaluates to a value. A primitive expression is a single evaluation step: you either look up the value of a name or take the literal value. For example, numbers and strings are all primitive expressions. Call expressions are simply expressions that involve a call to some function using pair of parentheses. Call expressions call some function with arguments and return some value from them.

EVALUATION OF A CALL EXPRESSION

The evaluation of a function call expression proceeds as follows:

1. First python evaluates the operator and all the operands.

2. Next python applies the function which is the value of the operator to the arguments which are the values of the operands.

FUNCTION CALL STACK

To understand how Python performs function calls, consider a collection of data known as a stack. Stacks can be thought of like a pile of dishes. When you add a dish to the pile, you place it on the top. Similarly, when you remove a dish from the pile, you take it from the top. Stacks are known as last-in, first-out (LIFO) data structures— the last item pushed or placed) onto the stack is the first item popped or removed from the stack.

Similarly, the function-call stack supports the function call/return mechanism by pushing and popping values local to a function. Eventually, each function must return program control to the point at which it was called. For each function call, the interpreter pushes an entry called a stack frame (or an activation record) onto the stack. This entry contains the return location that the called function needs so it can return

control to its caller. When the function finishes executing, the interpreter pops the function's stack frame, and control transfers to the return location that was popped.

ENVIRONMENT FRAMES

An environment is similar to a namespace, which is a set of bindings of variable names to values. Every time we encounter a variable name, we look up its value in the current environment. Note that we only lookup the value when we see/read the variable, and so we'll lookup the value of x only after we've already defined x. Otherwise this will throw an exception called a name error - since the name is not yet defined.

An environment frame is like a box that contains a set of bindings from variables names to values. An environment frame can "extend" another frame; that is, a frame can see all bindings of the frame it extends. We represent this by drawing an arrow from an environment frame to the frame it is extending.

Names may occur in multiple frames and are resolved to a value by starting in the current extension frame and following the arrows back to the extended frame. If the name cannot be resolved by reaching the global environment frame, then it can't be resolved, and thus we have a name error! Remember that the global environment is the only environment that extends nothing, and we therefore stop there.

HOW INTERPRETERS USE ENVIRONMENT FRAMES

1. Interpreters start off with the frame labeled global environment. This box starts out with bindings for all the built-in functions like +, abs, max, etc.

2. Interpreters have an active frame which is set initially to the global environment.

3. Interpreters evaluate each expression, one by one, by evaluating primitive expressions and resolving name bindings.

4. Each function call extends the active environment with a new frame.

When a function is called python first evaluates all the arguments as normal. Then, a new frame box is created as the new current frame. All the parameters are put into the frame box pointing to the argument values previously evaluated. Python then evaluates the body of the function in the current active frame. Once done, the active frame goes back to the frame from which the function call was made.

SCOPE RULES AND GLOBAL VARIABLES

A function may have a local variable name and it has local scope. A local variable name can be used only inside the function that defines it. The local variable name is only in scope only from its definition to the end of the function's block. The name goes out of scope when the function returns to its caller.

Names defined outside any function have global scope. Variables with global scope are known as global variables.

Identifiers with global scope can be used in a interactive session anywhere after they are defined.Y ou can access a global variable's

value inside a function, however, by default, you cannot modify a global variable in a function—since whenever you first assign a value to a variable within a function block, Python creates a new local variable.

To modify a global variable in a function's block, you must use a global statement to declare that the variable is defined in the global scope and not within the local function.

```
def modify_global_var():
    global x
    x = ‘local x'
    print('x printed from modify_global:', x)

>>> modify_global()
x printed from modify_global: local x

>>> x = ‘global x’
>>> modify_global()
x printed from modify_global: local x
```

HIGHER ORDER FUNCTIONS

A Higher Order Function is a function which takes other functions as arguments or returns a function (or both). Suppose we'd like to evaluate one function on each natural number from 1 to n and print the result as we

go along. For example, say we want to square every number from 1 up to n. We can generalize functions of this form into something more convenient. When we pass in the number n as an argument we could also pass in the particular function at the same time. To do that, we define a higher order function. The function takes in the function you want to apply to each element as an argument, and applies it to each of the n natural numbers starting from 1.

```
def square(x):
    return x * x

def apply(func, n):
    result = func(1)
    for i in range (1,n+1):
        result = func(i)
        print(result, end = " ")

>>> apply(square,10)
1 4 9 16 25 36 49 64 81 100
```

Python functions can also return functions. Sometimes we wish to write a function so that given some set of arguments returns a function that will do something with those arguments when it is called. The key here is that your (higher order) function is designed

to return a function and not evaluate anything yet.

LAMBDA EXPRESSIONS

For simple functions it is common practice in Python to use a lambda expression (or simply a lambda) to define the function inline where it's needed— typically as it's passed to another function.

A lambda expression is an anonymous function—that is, a function without a name.

```
>>> apply(lambda x: x*x , 5)
1 4 9 16 25
```

Here we pass to the function apply a lambda expression, rather than the function name square.

A lambda begins with the lambda keyword followed by a comma-separated parameter list, a colon (:) and an expression. In this case, the parameter list has one parameter named x. A lambda implicitly returns its expression's value, which in this case is the square of x. Note that any number of arguments and any simple expression

involving those arguments may be used within the lambda expression.

DISPATCH FUNCTIONS

A dispatch function is a classic example of a higher order function that returns a function. Dispatch functions takes several arguments and links a function to each one through a registration process. A dispatch function gives a programmer the flexibility to use a single named function to operate differently based on single special argument, which we call an option. Each option is associated with a different function used to process the values(s).

The following simple example shows how a dispatch function can be created with 2 options each associated with a unique functions. The dispatch function is designed to register each of the option associations and returns a single function with two arguments, one for the option and one for the value the option function should be applied to. In the following code note how a function is defined and then returned in the dispatch function.

```
def f_to_c(fahrenheit):
    return (fahrenheit - 32) * 5 / 9
def c_to_f(celsius):
    return (celsius) * 9 / 5 + 32

def dispatch_function(option1, f1,
                      option2, f2):
    """Takes in two options and two functions.
Returns a function that takes in an option and
value and calls either f1 or f2 depending on the
given option.

    >>> func_d = dispatch_function('c to f',
c_to_f, 'f to c', f_to_c)
    >>> func_d('c to f', 0)
    32.0
    >>> func_d('f to c', 68)
    20.0
    >>> func_d('blabl', 2)
    AssertionError
    """
    def func(option, value):
        assert option == option1 or
               option == option2
        if option == option1:
            return f1(value)
        else:
            return f2(value)
    return func
```

CHAPTER 2 EXERCISES

Homework #2: Download the the folder for the Lab 2 on Higher Order Functions. Extract the files in the folder if needed and open the index.html file in you browser. The Lab worksheet should be formatted with a Red

Banner on top. Turn in a file lab2.py with your answers to the Required Questions

LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 3: RECURSION

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

3.

RECURSION

BY FRED ANNEXSTEIN

In this module we will be covering the basics of using recursion in computational problem solving.

CHAPTER 3 OBJECTIVES

By the end of this module, you will be able to...
Understand what recursion is and how it is handled by the interpreter.
Identify the base case and the recursive step in a recursive function.
Correctly use recursion in problem solving.
Understand the mathematical significance of classical recursive functions, such as factorial, harmonic numbers.
Compare and contrast recursive and iterative solutions to problems.
Identify the contexts in which recursive solution are efficient or inefficient.

RECURSION

A recursive function is one that can call itself before it returns. Evaluating a call to a recursive function is not essentially different from evaluating a non-recursive function.

Recursion lends itself to an important problem-solving approach. This approach structures a solution with a base case and an inductive case. Base cases are typically trivial as they solve the simplest cases or directly without further processing. If you call the function satisfying a base case, it immediately returns with a result. If you call the function with the inductive case (more complex than a base case), the function will typically divides the problem into pieces. The pieces must be slightly simpler or a smaller version of the original problem. Since each problem piece resembles the original problem, the function calls a fresh copy of itself to work on the smaller problem piece(s) —this is called the recursion step. The recursion step executes while the original function call is still active and not yet complete. This situation can result in many more recursive calls as the function divides each new subproblem until a base case is reached.

For proper execution the recursion must eventually terminate, thus converging on a base case. When the function recognizes the

base case, it returns a result to the previous copy of the function. A sequence of returns ensues until the original function call returns the final result to the original caller - the main program. It is possible however to misuse recursion resulting in an infinite loop of recursive calls and thus never return.

Understanding and using recursion correctly is essential for any programmer. When applying recursion in problem solving, consider how a solution to a simpler version of the problem can be used to solve a more complex version. Remember to trust the recursion to work as designed: you can od this by assuming that your solution to the simpler problem works correctly without worrying about the details of how it accomplishes the task.

Begin your solution by thinking about what the answer would be in the simplest possible case(s). These will be your base cases - the stopping points for your recursive calls. Make sure to consider the possibility that you are missing base cases for this is a common way recursive solutions fail. You may also find it helpful to write the iterative

version first before constructing a recursive solution.

LIMIT ON RECURSION DEPTH

The depth of recursion is the number of times a recursion function calls itself. Python interpreters set a fixed limit on how many times one function can call itself to prevent infinite recursion. We can see the current limit and even reset that limit using a module called sys. Typically the default limit on recursion is 1000 calls. In the following example we show how to double that limit to 2000.

```
import sys
print(sys.getrecursionlimit())
sys.setrecursionlimit(2000)
print(sys.getrecursionlimit())
```

COUNTING UP AND DOWN WITH RECURSION

Here are two simple recursive functions, one that counts up to a number given as argument n, and the other counts down. Try

to see if you can understand and argue that these are correct implementations.

```
def countup(n):
    if n == 0:
        print(0, end = ' '))
    else:
        countup(n-1)
        print(n, end = ' ')

def countdown(n):
     if n == 0: print(0, end = ' ')
     else:
         print(n, end = ' ')
         countdown(n-1)

>>> countup(10)
0 1 2 3 4 5 6 7 8 9 10

>>> countdown(10)
10 9 8 7 6 5 4 3 2 1 0
```

One can argue that these functions are correct using the following type of argument. Let us consider the countup() function. We know the base case is when n=0 and so our function is correct on that input. Now we assume that our function countup works for all values up to k >=0. Let us show it is correct on input k+1. This is called the inductive step. In this case the function will execute the else clause, and since we have assumed countup(k) is correct we know that it will print out the numbers 0, 1, up to k. Finally it prints the final value k+1, and

terminates correctly. Try to construct a similar argument for the countdown function.

RECURSIVE FUNCTIONS FOR FACTORIAL AND FIBONACCI

You can arrive at a recursive factorial representation by observing that n factorial or n! can be written as:

```
n! = n · (n – 1)!
```

For example, 5! is equal to 5 · 4!, as in:

```
5! = 5 · (4 · 3 · 2 · 1)
5! = 5 · (4!)
```

We can define the function factorial recursively in python as follows:

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

We know by its definition that 0! is 1. So we choose n = 0 as our base case. The

recursive step also follows from the definition of factorial, i.e. n! = n * (n-1)!.

Let us turn now to the fibonacci function which is defined recursively as follows:

Fib(0) = Fib(1) = 1 , and
Fib(n) = Fib(n-1) + Fib(n-2), for all n>=2.
We can write this function is Python as follows.

```
def fib(n):
    if n >= 2: return fib(n-1) + fib(n-2)
    else:  return 1
```

In Python *list comprehensions* are a concise notation for creating list of values. List comprehensions use special for construction and can replace many multi-line for-loops that create new lists. More about lists will be covered in the next chapter. Here we construct two lists using the functions discussed above each within a list comprehension and display the results.

```
>>> one = [fac(i) for i in range(9)]
>>> two = [fib(i) for i in range(9)]
>>> one
```

```
[1, 1, 2, 6, 24, 120, 720, 5040, 40320]
>>> two
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Fibonacci is in a different class of recursive functions from Factorial. The fibonacci function exhibits a branching in their recursive calls, in this case in form of binary tree of recursion calls. This type of tree recursion can be very costly in execution time. This is demonstrate in the `lucas()` function from the first homework. As a challenge, try to determine the largest m for which `fib(m)` will terminate within few seconds.

ENVIRONMENTS TO EVALUATE RECURSIVE FUNCTIONS

Evaluating a call to a recursive function is not essentially different from evaluating a non-recursive function. Recall that Python implements function calls with a stack-frame containing information on local data. Each recursive function call gets its own stack-frame on the function-call stack. When a given call completes, the system pops the function's stack frame from the stack and

control returns to the caller, which is possibly another copy of the same function. So each recursive call extends the current environment of stack-frames adding any new local objects referenced.

RECURSIVE FUNCTION FOR DIGIT SUMS

Suppose that we want a function that will break a number into a collection of single digits, and then produce the sum of all the digits. To design such a function we can use two arithmetic operations to break up a given number into two. The mod operator %10 will produce the last or low order digit. Integer division by 10 (using //10) will produce the number that has all but the low-order digit, which is dropped.

The logic used to design the `sumdigits()` function will be to check the base case for when the number is just a single digit. If the number does not have a single digit, then we can recursively call the function on the remaining digits and add the low-order digit to the result as follows:

```
def sumdigits(n):
    """
    >>>sumdigits(134)
    8
    >>> sumdigits(5679)
    27
    """
    if n < 10:  return n
    else:
        last = n % 10
        prefix = n //10
    return last + sumdigits(prefix)
```

THE HARMONIC SERIES

Consider the function that is the sum of the first n unit fractions.

H(n) = 1 + 1/2 + 1/3 + 1/4 + ... + 1/n

In mathematics this sum is called the nth harmonic number. As n grows large this series, called the harmonic series, is divergent — that is, the infinite harmonic series grows without bound. The name harmonic series derives from the concept of overtones in music.

The difference between the nth harmonic number and the natural logarithm of n (denoted ln n) converges to a small

constant called the Euler–Mascheroni constant = 0.57721…..

Here is recursive code to test this difference and watch it converge.

```
def harmonic(n):
    if n>1:  return 1/n + harmonic(n-1)
    else: return 1

import math
for n in range(1,1000):
    print(n, ':',  harmonic(n) - math.log(n))
```

CHAPTER 3 EXERCISES

1. Run the example above for the harmonic series. Change the number of iterations from 1000 to 5000 and run it again. What happens and why?

2. Fix the error above to find differences up to the term harmonic(5000) - log(5000).

3. After 5000 iterations how close are last two differences, that is what is the value of [harm(5001) -log(5001)] - [harm(5000) - log(5000)]. ?

4. Lab 3 on Recursion. Extract the files in the Canvas folder if needed, and open the index.html file in your browser. The Lab sheet should be formatted with a Red Banner on top. Complete the lab by turning in lab3.py with your answers to all the required questions.

INTRODUCING AI-DS-ML

PYTHON PROGRAMMING

CHAPTER 4
LISTS AND SEQUENCES

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

4. LISTS AND SEQUENCES

BY FRED ANNEXSTEIN

CHAPTER 4 OVERVIEW

In this chapter we will be covering how data sequences are created using lists and tuples and processed in python. We abstract away the details of these data operations using the sequence interface. We define and implement several functional operations on sequences, and design their code implementations using both iterative and recursive problem solving and programming techniques.

CHAPTER 4 OBJECTIVES

By the end of this module, you will be able to...

- Understand and apply the role, syntax, and abstractions of lists, tuples, and sequences.
- Apply recursive problem solving when processing lists and sequences.
- Apply the concept of a programming interface to programming with sequences.

- Practice the use of generalized for loops, list comprehension, and list slicing.
- Implement the functional operations of map, filter, and frequency to sequences.

SEQUENCES

It is important when processing data to maintain and utilize ordered collections. The term sequence refers to an ordered collection of values. The sequence is a powerful, fundamental abstraction in computer science. Sequences are not instances of a particular built-in type or abstract data representation, but instead a collection of behaviors that are shared among several different types of data. That is, there are many kinds of sequences, but they all share common behavior. In particular, a sequence has a finite length. An empty sequence has length 0. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

LISTS

Python includes several native data types that are sequences, the most important of which is the list. A list is a sequence that can have arbitrary, but finite length.

Lists have a large set of built-in behaviors, along with specific syntax to express those behaviors. We have already seen the list literal, which evaluates to a list instance, as well as an element selection expression that evaluates to a value in the list. The built-in `len` function returns the length of a sequence. Below, digits is a list with four elements. The elements in a sequence can be accessed with 0-based indexing. The element at index 3 of digits is -1.

```
>>> digits = [1, 2, 8, -1]
>>> len(digits)
4
>>> digits[3]
-1
```

LISTS ARE MUTABLE

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
>>> digits[1] = 5
>>> digits
[ 1, 5, 8, -1]
```

The element in position 1 of digits, which used to be 2, is now updated to 5.

FOR AND *IN* CONSTRUCTS

Python's *for* and *in* constructs are extremely useful when working with lists and sequences. The *for* construct -- for var in list -- is an easy way to look at each element in a list (or other iterable collection). Do not add or remove from the list during iteration.

```
squares = [1, 4, 9, 16]
sum = 0
for num in squares:
    sum += num
print(sum)  ## 30
```

The *in* construct is used as an easy way to test if an element appears in a list (or other iterable collection) by returning True/False.

```
list = ['larry', 'curly', 'moe']
if 'curly' in list:
      print(‘NaNaNaNa’)
```

The for/in constructs are very commonly used in Python code and so you should just memorize their syntax. You can also use for/in to work on a string. The string acts like a list of its chars, so

```
# prints all the chars in a string.
for ch in s:
    print(ch)
```

LIST SLICES

Slices work on lists just as with strings, and can also be used to change sub-parts of the list.

```
>>> list = ['a', 'b', 'c', 'd']
>>> list[1:-1]
```

```
['b', 'c']
>>> list[0:2] = ‘z'
>>> list
['z', 'c', ‘d']
```

APPENDING TO A LIST WITH +=

We will start with an empty list, and then use a for statement and += operator to append the values 1 to 10 —note the list grows dynamically to accommodate each item.

```
a_list=[]
for i in range(1,11):
    a_list += [10*i]

>>> a_list
[10, 20, 30, 40, 50, 60, 70, 80, 90,
100]
```

TRAVERSING A LIST

The most common way to traverse the elements of a list is with a for loop. Here will will use a for loop over a range value to access list indices and list values.

```
a_list=[]
for i in range(1,11):
    a_list += [10*i]

for i in range(len(a_list)):
    print(i, a_list[i])
```

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to n−1, where n is the length of the list. Each time through the loop i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value. A for loop over an empty list never runs the body.

```
for x in []:
    print('This never runs.’)
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol
le Veq'], [1, 2, 3]]
```

TUPLES

Unlike lists, tuples are an immutable data type. However, similar to lists, tuples can store and access heterogeneous data. A tuple's length is its number of elements and cannot change during program execution.

To create an empty tuple, use empty parentheses: Recall that you can pack a tuple by separating its values with commas: When you output a tuple, Python always displays its contents in parentheses. You may surround a tuple's comma-separated list of values with optional parentheses:

ACCESSING TUPLE ELEMENTS

A tuple's elements, though related, are often of multiple types. Usually, you do not iterate over them. Rather, you access each individually. Like list indices, tuple indices

start at 0. The following code creates time_tuple representing an hour, minute and second, displays the tuple, then uses its elements to calculate the number of seconds since midnight—note that we perform a different operation with each value in the tuple:

```
>>> student_tuple = ()
>>> student_tuple
()
>>> len(student_tuple)
0
>>> student_tuple = 'John', 'Green',
3.3
>>> student_tuple
('John', 'Green', 3.3)
>>> len(student_tuple)
3
```

LIST METHODS- APPEND AND EXTEND

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
```

```
['a', 'b', 'c', 'd']
```
`extend` takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```
This example leaves `t2` unmodified.

SORT METHOD

sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

Most list methods are not pure; they modify the list and return `None`. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

MAP, FILTER, AND REDUCE

To add up all the numbers in a list t, you can use a loop like this:

```
def add_all(t):
  total = 0
  for x in t:
    total += x
return total
```

Note that total is initialized to 0. Each time through the loop, x gets one element from the list. The += operator provides a short way to update a variable.

As the loop runs, total accumulates the sum of the elements; a variable used this way is sometimes called an accumulator. Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

An operation like this that combines a sequence of elements into a single value is called a reduce function or a reduction.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings.

```
def capitalize_all(t):
  result = []
  for s in t:
    result.append(s.capitalize())
  return result
```

The variable result is initialized with an empty list; each time through the loop, we append the next element. So result is another kind of accumulator. An operation like capitalize_all is sometimes called a map because it “maps” a function (in this case the method capitalize) onto each of the elements in a sequence. Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings.

```
def only_upper(t):
  result = []
```

```
  for s in t:
        if s.isupper():
            result.append(s)
  return res
```

The function isupper is a builtin string method that returns True if the string contains only upper case letters.

An operation like only_upper is called a filter because it selects some of the elements and filters out the others. Most common list operations can be expressed as a combination of map, filter and reduce.

VARIABLE LENGTH ARGUMENT LISTS

We define an average function that can receive any number of arguments. The parameter name args is used by convention, but you may use any identifier. If the function has multiple parameters, the *args parameter must be the rightmost parameter. Now, let us call average several times with arbitrary argument lists of different lengths: To calculate the average, divide the sum of the args tuple's elements (returned by built-in function sum) by the tuple's number of elements (returned by built-in function len).

```
def average(*args):
  return sum(args) / len(args)
```

Note in our average function definition that if the length of args is 0, a ZeroDivisionError occurs.

The parameter name args is used by convention, but you may use any identifier. If the function has multiple parameters, the *args parameter must be the rightmost parameter. Now, let's call average several times with arbitrary argument lists of different lengths:

```
>>> average(5, 10)
7.5
>>> average(5, 10, 15)
10.0
>>> average(5, 10, 15, 20)
12.5
```

THE * OPERATOR

The * is a special operator in python. When the * operator is applied to an iterable argument in a function call, the effect is to unpack its elements. The following code

creates a five-element grades list, then uses the expression *grades to unpack its elements as average's arguments:

```
>>> grades = [88, 75, 96, 55, 83]
>>> average(*grades)
79.4
```

The call shown above is equivalent to average(88, 75, 96, 55, 83).

LIST COMPREHENSIONS

Many sequence processing operations can be expressed by evaluating a fixed expression for each element in a sequence and collecting the resulting values in a result sequence. In Python, a list comprehension is an expression that performs such a computation.

```
>>> odds = [1, 3, 5, 7, 9]
>>> [x*10 for x in odds]
[10, 30, 50, 70, 90]
```

The `for` keyword above is not part of a `for` statement, but instead part of a list comprehension because it is contained within square brackets. The sub-expression `x+1` is evaluated with `x` bound to

each element of `odds` in turn, and the resulting values are collected into a list.

Another common sequence processing operation is to select a subset of values that satisfy some condition. List comprehensions can also express this pattern, for instance selecting all elements of `odds` that evenly divide `25`.

```
>>> [x for x in odds if 25 % x == 0]
[1, 5]
```

EXERCISES

Exercise 1. Write a function called nested_sum that takes a list of lists of integers and adds up the elements from all of the nested lists. For example:

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

Exercise 2. Write a function called cumu that takes a list of numbers and returns the cumulative sum; that is, a new list where the ith element is the sum of the

first i+1 elements from the original list. For example:

```
>>> t = [1, 2, 3]
>>> cumu(t)
[1, 3, 6]
```

Exercise 3. Write a function called middle that takes a list and returns a new list that contains all but the first and last elements. For example:

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

Exercise 4. Write a function called chop that takes a list, modifies it by removing the first and last elements, and returns None. For example:

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

Exercise 5. Write a function called is_sorted that takes a list as a parameter and returns True if the list is

sorted in ascending order
and False otherwise. For example:

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

Exercise 6. Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called is_anagram that takes two strings and returns True if they are anagrams.

Exercise 7. Write a function called has_duplicates that takes a list and returns True if there is any element that appears more than once. It should not modify the original list.

Exercise 8. This exercise pertains to the so-called Birthday Paradox, which you can read about at http://en.wikipedia.org/wiki/Birthday_paradox.

If there are 23 students in your class, what are the chances that two of them have the same birthday? You can estimate this

probability by generating random samples of 23 birthdays and checking for matches. Hint: you can generate random birthdays with the randint function in the random module.

PROGRAMMING CHALLENGE #2

Before attempting this challenge you should have some background knowledge of what the outcome of Newton's method does. We will implement the method here.

First consider the idea of an iterative numerical method. For example, here is an iterative methods to calculate pi. Let

$P(1) = 3.0$, and for $n>0$

$P(n + 1) = P(n) + \sin(P(n))$

...where P(n) is the approximation of pi at iteration n. Given a first estimate P(1) of pi as 3.0, this iteration converges to an approximation of pi correct to as many digits of precision that you are given when you call the sin function.

P(1) = 3.0

P(2) = P(1)+ sin(3.0) = 3.1411200...

P(3) = 3.14112 + sin(3.14112) = 3.141592654…

And so on.

After the second iteration, the error is very small (about 1.8*10^(-11)). And, on the next iteration, the error drops to about 10^(-32). Why does it this work so darn well?

This algorithm is a particular case of a method called fixed point iteration, and is used to find solutions to equations of the form:

x = f(x) [1]

In this particular case, we have f(x) = x + sin(x), and, as sin(n*pi) = 0 for any integer n, any multiple of pi is a solution of that equation.

An equation like [1] can sometimes (often?) be solved by iterating the formula:

x[n+1] = f(x[n])

where x[n] is the nth approximation of the solution. There is a famous folk theorem which states that the method will converge to the solution if the absolute value of the derivative f'(x) is less than some number L < 1 in some interval containing the solution, and if you start the iteration with a value in that interval. The smaller the derivative is, the faster the sequence will converge. In the case of f(x) = x + sin(x), we have: f'(x) = 1 + cos(x) and, if x is close to pi, cos(x) is close to -1, and f'(x) is close to 0, which explains why the method converges rapidly. We can also notice that f"(x), the second derivative, is -sin(x), which is 0 at the root. This means that you can start in a relatively large interval about pi, and still get a very fast convergence.

We can see this as a slight modification of Newton's method applied to the solution of the equation sin(x) = 0. Newton's method gives the formula: x[n+1] = x[n] - sin(x[n])/ cos(x[n] and, as cos(x[n]) is close to -1, you get your formula by replacing cos(x[n]) by -1 in the above formula.

Programming Challenge #2 Deliverables:

0. Source code that will run two functions fixed_point_iteration() and newton's method newton_find_zero(). See Sample Runs below.

1. Determine if Newton's method or the fixed point method is better for computing pi (up to 15 digits of accuracy) by counting how many iterations each needs to converge.

2. Determine if Newton's method or the fixed point method is better for computing the Dottie number (0.739085...), which is the unique, real number fixed point of the cos() function.

3. Include comments that discuss your successes and analysis of this problem.

You should write a function with an f-arg to compute an iterative fixed point iteration as follows.

```
def fixed_point_iteration(f, x=1.0):
    step = 0
    while not approx_fixed_point(f, x):
        x = f(x)
        step += 1
    return x, step

“””
```

```
“””
def approx_fixed_point(f, x) should
return True if and only if f(x) is
very close (distance < 1e-15) to x.
“””
```

Here is a set of sample runs which you should convert to doctests:

```
>>> from math import sin,cos
>>> fixed_point_iteration(lambda x:
sin(x) + x, 3.0)
(3.141592653589793, 3)

>>> newton_find_zero(lambda x:
sin(x) , lambda x: cos(x), 3.0)
(3.141592653589793, 3)

>>> print(fixed_point_iteration(lambda
x: cos(x), 1.0)
(0.7390851332151611, 86)
# Fixed point defining dottie number
>>> newton_find_zero(lambda x: cos(x)
- x , lambda x: -sin(x)-1, 1.0)
(0.7390851332151606, 7)
# Newtons's zero giving dottie
```

LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 5 DICTIONARIES AND HASHING

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

5. DICTIONARIES AND HASHING

BY FRED ANNEXSTEIN, PHD

CHAPTER 5 OVERVIEW

In this module we will be covering the important concept of searchable hash tables used for efficient data lookups. We show how to use the built-in python datatype called a dictionary to construct them. Dictionaries are one of Python's most valued features, and they are the key datatype for many efficient and elegant algorithms. We show how to use hash tables for frequency tables and for natural language models and we construct an AI chatbot based on such tables.

CHAPTER 5 OBJECTIVES

By the end of this module, you will be able to...

- Understand the syntax and methods of python dictionaries.
- Apply and implement tables such as frequency, histograms, and hash tables from python dictionaries.
- Construct a recursive function that uses memoization.
- Apply successor table dictionaries in the construction of a natural language model
- Implement a chatbot application using dictionary-based successor table.

DICTIONARY SYNTAX

Consider storing a collection of key/value pairs. Keys are used to locate table records which contain the values of interest. Python's efficient key/value table structure is called a dictionary or simply a ‘dict’. The contents of a dict can be written as a series of key:value pairs within braces { } surrounding, as follows:

```
dict1 = {key1:value1, key2:value2, ... }
```

We contract an "empty dict" with just an empty pair of curly braces.

```
dict1 = {}
```

Looking up or setting a value in a dict uses square brackets, for example, the following statement looks up the value associated with the key ‘foo'.

```
dict1[‘foo’]
```

Strings, numbers, and tuples all work as keys, and an object of any type can be a value. Other types may or may not work correctly as keys (strings and tuples work cleanly since they are immutable).

Looking up a value which is not in the dict throws a KeyError. You can use "in" operator to check if the key is in the dict, or use statement `dict.get(key)` which returns the value paired with key, or returns None if key is not present. Adding another argument will allow you to specify what value to return in the case of key not-found.

Let us look at a code block which we use a dictionary to build a hash table, starting with the the empty dict {}, and storing key/value pairs adding one at a time to the table:

```
dict1 = {}
dict['a'] = 'alpha'
dict['g'] = 'gamma'
dict['o'] = 'omega'

>>> dict1
{'a': 'alpha', 'g': 'gamma', 'o': 'omega'}

>>> dict1['a']
‘alpha’

>>> dict1['b']
Traceback (most recent call last):
KeyError: ‘b'

>>> dict1[‘b'] = 6

>>> dict1[‘b'])
6
>>> 'b' in dict1
True
```

HISTOGRAMS

Dictionaries are useful for computing the frequency of values in a collection. Here is an example that will work with any sequence type including strings.

```
def frequencies(c):
   freq = {}
   for i in c:
       if i not in freq:
           freq[i] = 1
       else: freq[i] += 1
   return freq

>>> frequencies('hello')
{'h': 1, 'e': 1, 'l': 2, 'o': 1}
```

The first line of the function creates an empty dictionary. The for loop traverses the string. Each time through the loop, if the character is not in the dictionary, we create a new item with key and the initial value 1, since we have now seen this letter once. If i is already in the dictionary, then we increment the counter value.

FOR LOOPS

You can use a for-loop with a dictionary, and we therefore say that a dictionary is iterable. This iteration traverses the keys of the dictionary. For example, print_dict(d) prints each

key and the corresponding value in the dictionary d.

```
def print_dict(d):
   for c in d:
       print(c,':' d[c])

>>> h = frequencies('rabbitt')
>>> print_dict(h)
r:1  a:1  b:2  i:1  t:2
```

MEMOIZATION

Recall the recursive definition of the Fibonacci function from Chapter 3. We saw that the tree recursion meant that the function would branch so much that the function will not finish for even modest argument values.

One solution to this problem is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored in the dictionary for later use is called a memo. Here is a "memoized" version of Fibonacci function. We use name 'known' for a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

```
known = {0:0, 1:1}

def fib(n):
    if n in known:
        return known[n]
    result = fib(n-1) + fib(n-2)
    known[n] = result
    return result

>>> for i in range(1,6):
        print(i,':',fib(i), end='  ')

1 : 1   2 : 1   3 : 2   4 : 3   5 : 5
```

Whenever the fib function is called, it checks the known dictionary. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it. If you run this version of fib and compare it with the original recursive version, you will find that it is much faster for large arguments.

SUCCESSOR TABLES

We now consider the problem of building a hash table for a natural language model, and use that model to create a bot for Twitter. The natural language model we use is called the bigram word model. That is, for every word in a text file or corpus of files we will store all possible successor words. These pairs will be stored in a "successor table" implemented using Python dictionaries.

Once we are done building the successor table we can generate a sentence or 'tweet' by starting with a (random) word and then randomly choosing a word from the successor list. Then we repeat the processes to look up the next successor word, and so on. This process eventually will terminate in a period (".") or other terminal character.

The word successor table is just hash table and is implemented using an ordinary Python dictionary. The keys in this dictionary are (predecessor) words, and the values are lists of all successor words found in the corpus.

Here is the definition of a function to build_successors_table. The input argument tokens is a list of words corresponding to a given corpus text that is split into “tokens" or individual words. The output is a successors table. (By default, we assign the first word is a successor to "."). See the example below.

```
def build_successors_table(tokens):
    table = {}
    prev = '.'
    for word in tokens:
        if prev not in table:
            table[prev] = []
        table[prev] += word
        prev = word
    return table
```

GENERATING RANDOM SENTENCES

Let us generate some sentences at random simply by using the successor table. Suppose we are given some starting word. We can look up this word in our table to find its list of successors, and then randomly select a word from this list to be the next word in the sentence. Then we just repeat the process until we reach some ending punctuation.

To randomly select from a list, first make sure you import the Python random library with `import random` and then use the expression `random.choice(my_list))`

MARKOV CHAINS

In our successor table example above, we used a list of words as our values. In our Twitter application we treated each word equally by making a random choice when we generated new sentences. We can get more powerful models by allowing bias which associates a belief likelihood or probability weight to each element of the list of values. These weights bias the possible outcomes to be more probable choices. This is the idea of a Markov chain - a table in which the set of values is assigned a likelihood or probability.

Here is a simple example script for a Markov Chain modeling three states of the daily weather. We construct a Markov chain by associating a successor probability with each possible day state. We represent the weights by enhancing our dictionary where we let table values be lists of pairs stored as tuples. Each tuple represents a daily weather state along with its probability as successor state. Note that in our example below when it is 'Sunny' there is equal 1/3 chance that the next day will be any state, but when it is 'Raining' the chance of rain the next day is 3/5.

```
import random
s,c,r = 'Sun', 'Clouds', 'Rain'
weather = [s,c,r]
weather_table = { s : [(s,1/3),(c,1/3),(r,1/3)],
                  c : [(s,1/4), (c,2/4), (r,2/4)],
                  r : [(s, 1/5), (c,2/5), (r, 3/5)]}
```

Here weather only depends on the previous day and the probabilities associated with that weather state is stored as a list of tuples in the table. Here is a randomly generated 10-day forecast based on this Markov Chain model.

```
nextday = random.choice(weather)
for day in range(10):
   weatherlst = weather_table[nextday]
   print(s,c,r)
   probs = [w[1] for w in weatherlst]
   print(probs)
   nextday = random.choices(weather, probs, k=1)[0]
   print("Day", day, ":  ", nextday)

Sun Clouds Rain
[0.25, 0.5, 0.25]
Day 0 :   Clouds
```

```
Sun Clouds Rain
[0.25, 0.5, 0.25]
Day 1 :   Clouds
Sun Clouds Rain
[0.25, 0.5, 0.25]
Day 2 :   Rain
Sun Clouds Rain
[0.2, 0.2, 0.6]
Day 3 :   Sun
Sun Clouds Rain
[0.333, 0.333, 0.333]
Day 4 :   Sun
```

Note that we are passing to the **`random.choice`** function an optional parameter representing Markov Chain weights which we name **`probs`**. These probabilities were extracted or projected from the weather_table using a list comprehension to extract just a list of numbers to be used as an argument.

CHAPTER EXERCISES

Exercise 1. Write a function that reads the words in the file words.txt and stores them as keys in a dictionary. It doesn't matter what the values are. Then you can use the in operator as a fast way to check whether a string is in the dictionary.

Exercise 2. Memoize the Ackermann function, see the definition at https://en.wikipedia.org/wiki/Ackermann_function

Exercise 3. Use a dictionary to write a fast predicate function called **`has_duplicates(seq)`** which takes an argument that is a sequence, and

determines whether or not there are any duplicate values in the sequence.

Exercise 4. The “rank” of a word is its position in a list of words sorted by frequency: the most common word has rank 1, the second most common has rank 2, etc.

Zipf’s law describes a relationship between the ranks and frequencies of words in natural languages (http://en.wikipedia.org/wiki/Zipf's_law). Specifically, it predicts that the frequency, f, of the word with rank r is:

$$f = c\ r^{-s}$$

where s and c are parameters that depend on the language and the text. If you take the logarithm of both sides of this equation, you get:

$$\log f = \log c - s \log r$$

So if you plot log f versus log r, you should get a straight line with slope –s and intercept log c.

Write a program that reads a text from a file, counts word frequencies, and prints one line for each word, in descending order of frequency, with log f and log r. Use the graphing program of your choice to plot the results and check whether they form a straight line. Can you estimate the value of s?

LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 6
LINKED LISTS + TREES

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

6.

LINKED LISTS AND TREES

BY FRED ANNEXSTEIN

CHAPTER 6 OVERVIEW

In this module we will be covering the abstract data types of linked lists and trees. We will examine the creation and processing of these objects using functional programming style. We will apply out tree implementations to the solution of several programming problems.

CHAPTER 6 OBJECTIVES

By the end of this module, you will be able to...

- Understand the notion of an abstract data type using the example abstractions of linked lists and trees.
- Understand a functional approach to the implementation of creating and processing linked lists and trees.
- Understand how to use recursion to effectively process linked lists and trees.
- Understand how to use linked lists and trees as a data structure underlying an immutable sequence.

LINKED LISTS

Python has many built-in types of sequences: lists, ranges, and strings, to name a

few. We construct our own type of sequence called a linked list. A linked list is a simple type of sequence that is composed of multiple links that are connected.

The key idea for the link abstraction is that each link is a pair where the first element is an item-value in the linked list, and the second element is another link. Links can be also be empty, that is a special value marking the end of the linked list.

CONSTRUCTORS

We design a function to construct a linked link as follows.

```
empty = 'empty'

def link(first, rest=empty):
    return [first, rest]
```

This function constructs a linked list with a first element and next element, which is also a link list or defaults to the empty value for the empty linked list.

SELECTORS

We design two function to select elements from a linked link as follows.

```
def first(s):
    return s[0]

def rest(s):
    return s[1]
```

A linked list is a pair containing the first element of the sequence (in this case 1) and the rest of the sequence (in this case a representation of 2, 3, 4). The second element is also a linked list. The rest of the inner-most linked list containing only 4 is 'empty', a value that represents an empty linked list. Linked lists have recursive structure: the rest of a linked list is a linked list or 'empty'. We can define an abstract data representation to validate, construct, and select the components of linked lists.

```
>>> alpha = link(‘A’, link(‘B’,
link(‘C’, link(‘D’))))

>>> first(alpha)
'A'

>>> rest(alpha)
['B', ['C', ['D', 'empty']]]

>>> first(rest(alpha))
'B'
```

LINKED LIST SEQUENCE INTERFACE

The linked list can store a sequence of values in order, but we have not yet shown that it satisfies the sequence abstraction. Using the abstract data representation we have defined, we can implement the two behaviors that characterize a sequence: length and element selection.

```
def len_link(s):
    length = 0
    while s != empty:
       s, length = rest(s), length 1
    return length
```

```
def getitem_link(s, i):
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

```
>>> len_link(alpha)
4

>>> getitem_link(alpha, 3)
'D'
```

Now, we can manipulate a linked list just as we can with any sequence using these interface functions.

RECURSIVE IMPLEMENTATION

Both len_link and getitem_link are iterative. They move through each layer of nested pair until the end of the list (in len_link) or the desired element (in getitem_link) is reached. We can also implement length and element selection using recursion.

```
def len_recursive(s):
    if s == empty: return 0
    return 1 + len_recursive(rest(s))

def getitem_recursive(s, i):
    if i == 0: return first(s)
    return getitem_recursive(rest(s),
                             i - 1)
>>> len_recursive(alpha)
4
>>> getitem_recursive(alpha,3)
'D'
```

SUMMING ELEMENTS

Let us write a function that takes in a linked list `lst` and a function `fn` which is applied to each number in `lst` and returns the sum.

```
# Recursive Solution
def sum_linked_list_r(lst, fn):
    if lst == empty:
        return 0
    return fn(first(lst)) +
sum_linked_list(rest(lst), fn)

# Iterative Solution
def sum_linked_list_i(lst, fn):
    sum = 0
    while lst != empty:
        sum += fn(first(lst))
        lst = rest(lst)
    return sum

>>> sum_linked_list_r(alpha, ord)
266

>>> sum_linked_list_r(alpha,
            lambda x: ord(x)-ord('A')
6
```

In the last example we use Python ord() function which returns the Unicode code from a given character. This function accepts a string of unit length as an argument and returns the Unicode equivalence of the passed argument. In other words, given a string of length 1, the ord() function returns an

integer representing the Unicode code point of the character when an argument is a Unicode object, or the value of the byte when the argument is an 8-bit string. For example, ord(‘A’) returns the integer 65, ord(‘a’) returns the integer 97, ord(‘€’) (Euro sign) returns 8364. The ord function is the inverse of the chr() functions for 8-bit characters and of unichr() for Unicode objects. Character’s code point must be in the range [0..65535] inclusive. We can print math symbols since they reside starting at 8704 code point.

```
>>> for i in range(8704,8854):
    print(chr(i), end='  ')
∀  ∁  ∂  ∃  ∄  ∅  ∆  ∇  ∈  ∉  ∊  ∋  ∌
∍  ∎  ∏  ∐  ∑  −  ∓  ∔  ∕  ∖  ∗  ∘  ∙
√  ∛  ∜  ∝  ∞  ∟  ∠  ∡  ∢  ∣  ∤  ∥  ∦
∧  ∨  ∩  ∪  ∫  ∬  ∭  ∮  ∯  ∰  ∱  ∲
∳  ∴  ∵  ∶  ∷  ∸  ∹  ∺  ∻  ∼  ∽  ∾  ∿
≀  ≁  ≂  ≃  ≄  ≅  ≆  ≇  ≈  ≉  ≊  ≋  ≌
≍  ≎  ≏  ≐  ≑  ≒  ≓  ≔  ≕  ≖  ≗  ≘  ≙
≚  ≛  ≜  ≝  ≞  ≟  ≠  ≡  ≢  ≣  ≤  ≥  ≦
≧  ≨  ≩  ≪  ≫  ≬  ≭  ≮  ≯  ≰  ≱  ≲  ≳
≴  ≵  ≶  ≷  ≸  ≹  ≺  ≻  ≼  ≽  ≾  ≿  ⊀
⊁  ⊂  ⊃  ⊄  ⊅  ⊆  ⊇  ⊈  ⊉  ⊊  ⊋  ⊌  ⊍
⊎  ⊏  ⊐  ⊑  ⊒  ⊓  ⊔  ⊕
```

CREATING LINKED LISTS FROM LISTS

We now demonstrate a function that can construct a linked list from an ordinary list. We use a for loop to iterate through elements of the input list, and at each step we construct a new linked list by append the next element in turn as follows.

```
def list_to_link(lst):
    l_l = empty
    for x in lst:
        l_l = link(x,l_l)
    return l_l

>>> list_to_link(list(range(7)))
[[6, [5, [4, [3, [2, [1, [0,
‘empty']]]]]]]
```

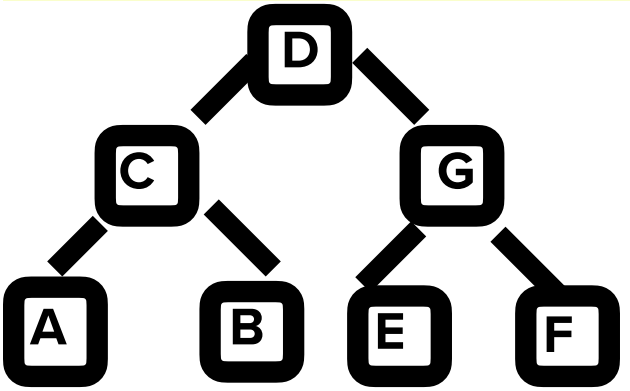
TREES

The tree is a generalization of linked list and is a fundamental data abstraction that imposes regularity on how hierarchical values are structured and processed. A tree has a root and a sequence of branches. Each branch of a tree is a tree. A tree with no branches is called a leaf. Any tree contained within a tree is called a sub-tree of that tree (such as a branch of a branch).

The root of each sub-tree of a tree is called a node in that tree.

The data abstraction for a tree consists of collection of functions including a constructor and selectors. We begin with a simplified version.

```
def tree(root_obj, branches=[]):
  return [root_obj] + list(branches)

def root(t):
  return t[0]

def branches(t):
  return t[1:]

def is_leaf(t):
  return not branches(t)


def add_branch(t,x):
  return tree(root(t),branches(t) +
[tree(x)])

alpha_tree =
  tree('D', [
   tree('C', [tree('A'),tree('B')]),
   tree('G', [tree('E'),tree('F')])])
```



TRAVERSE TREE

To traverse a tree is to visit each node of the tree exactly once. Here is a function to do just that.

```
def traverse(t):
  print(root(t))
  if not is_leaf(t):
    for b in branches(t):
         traverse(b)

>>> traverse(alpha_tree)
D  C  A  B  G  E  F
```

When a node is first reached the function prints out the root value. It then checks if the node is a leaf node (this is the base case). If it is

not a leaf, a recursive call is made on each tree in the list of branches. Traverse always visits nodes in a depth-first order.

If we want to reach a random leaf of the tree we can do a random walk as follows. At each node, check if it is a leaf. If it is not we can choose a random branch to continue the walk. We use the randrange function from the standard random library to choose a random branch as we go down the tree.

```
from random import randrange

def random_walk(t):
  print(root(t), end= ' ')
  if not is_leaf(t):
    random_walk(branches(t)
        [randrange(len(branches(t)))])

>>> random_walk(alpha_tree)
D C B

>>> random_walk(alpha_tree)
D G E

>>>random_walk(alpha_tree)
D G F
```

FIBONACCI TREE

The following function builds and returns a binary tree that is constructed out of the

fibonacci sequence. A binary tree has exactly two branches from each non-leaf node. The function works recursively. The base case if when the value n is 1 or 0, and the function returns the single leaf with value 1. Otherwise, the function constructs a tree from the two sub-trees fibtree(n-1) and fibtree(n-2), setting them as two branches from root. The value stored in the root is simply the sum of the roots of the two branches.

```
def fibtree(n):
  if n ==1 or n==0:
    return tree(1)
  else:
    t1= fibtree(n-1)
    t2= fibtree(n-2)
    return tree(root(t1) + root(t2),
[t1] + [t2])

>>> f= fibtree(5)

>>> root(f)
8

>>> traverse(f)
8  5  3  2  1  1  1  2  1  1  3  2  1
1  1

>>> random_walk(f)
8 5 2 1

>>> random_walk(f)
8 5 3 1
```

```
>>> random_walk(f)
8 3 1

>>> random_walk(f)
8 3 2 1
```

CHAPTER 6 EXERCISES

Question 1: is_sorted

Implement the is_sorted(lst) function, which returns True if the linked list lst is sorted in increasing from left to right. If two adjacent elements are equal, the linked list is still considered sorted.

Question 2: Interleave

Write interleave(s0, s1), which takes two linked lists and produces a new linked list with elements of s0 and s1 interleaved. In other words, the resulting list should have the first element of the s0, the first element of s1, the second element of s0, the second element of s1, and so on.

If the two lists are not the same length, then the leftover elements of the longer list should still appear at the end.

Question 3: Height

Define the function height, which takes in a tree as an argument and returns the number of branches it takes start at the root node and reach the leaf that is farthest away from the root.

Note: given the definition of the height of a tree, if height is given a leaf, what should it return?

Question 4: Sprout leaves

Define a function sprout_leaves that, given a tree, t, and a list of values, vals, and produces a tree with every leaf having new children that contain each of the items in vals. Do not add new children to non-leaf nodes.

Question 5: DNA Sequence Matching

The mad scientist John Harvey Hilfinger has discovered a gene that compels people to enroll in CS2023. You may be afflicted!

A DNA sequence is represented as a linked list of elements `A`, `G`, `C` or `T`. This discovered gene has sequence `C A T C A T`. Write a function `has_2023_gene` that takes a DNA sequence and returns whether it contains the 2023 gene as a sub-sequence.

First, write a function `has_prefix` that takes two linked lists, `s` and `prefix`, and returns whether `s` starts with the elements of `prefix`.

Note that `prefix` may be larger than `s`, in which case the function should return `False`.

Question 6: Count Change (with Linked Lists!)

A set of coins makes change for `n` if the sum of the values of the coins is `n`. For example, if you have 1-cent, 2-cent and 4-cent coins, the following sets make change for `7`:

- 7 1-cent coins
- 5 1-cent, 1 2-cent coins
- 3 1-cent, 2 2-cent coins
- 3 1-cent, 1 4-cent coins
- 1 1-cent, 3 2-cent coins
- 1 1-cent, 1 2-cent, 1 4-cent coins

Thus, there are 6 ways to make change for `7`. Write a function `count_change` that takes a positive integer `n` and a linked list of the coin denominations and returns the number of ways to make change for `n` using these coins.

Question 7: Assume there is a function cons(a, b) that constructs a pair, and car(pair) and cdr(pair) returns the first and last element of that pair. For example,

```
>>> car(cons(3, 4))
3
>>> cdr(cons(3, 4))
4
```

Here is an implementation of cons which uses a function as a return value for a pair:

```
def cons(a, b):
    def pair(f):
        return f(a, b)
    return pair
```

Provide an implementation for the functions car and cdr.

LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 7: OBJECT ORIENTED PROGRAMMING

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

7.

OBJECT-ORIENTED PROGRAMMING

BY FRED ANNEXSTEIN

CHAPTER 7 OVERVIEW

In this module we will be covering an introduction to the style of object oriented programming using Python. Object-oriented design and object-oriented programming is quite popular in Python since the language natively supports the concepts of "classes, objects, attributes, and methods". Object-oriented design is a methodology that supports code reuse and the long-term maintenance of computer code.

CHAPTER 7 OBJECTIVES

By the end of this chapter, you will be able to...

- Understand the need and objectives of OOP.
- Understand the syntax for classes, objects, and methods.
- Apply the syntax of the language to correctly execute OOP codes.
- Understand inheritance and polymorphism as popularly used in Python for OOP.

DEFINING OOP

Object-oriented programming (OOP) is a methodology for organizing programs into units called classes and objects. OOP is an alternative design process for building data

abstraction and abstract data types. OOP classes create abstraction layers or barriers between the use and implementation of data types. Like the dispatch dictionaries we discussed in Chapter 2, objects respond to behavioral requests. Like mutable data structures, objects have a local state that is not directly accessible from the global environment. The Python object system provides convenient syntax to promote the use of these techniques for organizing programs. Much of this syntax is shared among other object-oriented programming languages.

OBJECT-BASED PROGRAMMING

The vast majority of object-oriented programming and design involves creating and using objects of existing classes. You have been doing this already when using built-in types like int, float, str, list, tuple, dict and set. Over the years, the Python open-source community has crafted an enormous number of valuable classes and packaged them into class libraries. This makes it easy for you to reuse existing classes. The vast majority of the classes you will need in programming are likely to be freely available in open-source libraries.

However, every competent programmer must master the design and implementation of custom classes. Unlike standard classes, these will be classes that you write yourselves.

The object system offers more than just convenience. It enables a new metaphor for designing programs in which independent agents interact within the computer. Each object bundles together local state and behavior in a way that abstracts the complexity of both.

Objects communicate with each other, and useful results are computed as a consequence of their interaction. Not only do objects pass messages, they also share behavior among other objects of the same type and inherit characteristics from related types. The paradigm of object-oriented programming has its own vocabulary that supports the object metaphor. We have seen that an object is a data value that has methods and attributes, accessible via dot notation. Every object also has a type, called its class. To create new types of data, we implement new classes.

CREATING YOUR OWN CUSTOM CLASSES

A class serves as a template for all objects whose type is that class. Every object is an instance of some particular class. The objects we have used so far all have built-in classes, but new user-defined classes can be created as well. A class definition specifies the attributes and methods shared among objects of that class. Classes that you create are new data

types. A class is a blueprint for a collection of objects. Classes can specify the use of data structures defined by the user. Users create new classes with the keyword class to define a block of code that keeps related things together.

Let's write a simple class starting with no attributes or methods. We instantiate an object or instance of the class Pokemon, and assign the object just created to the variable name poke.

```
class Pokemon:
    pass

poke = Pokemon()

>>> poke
<__main__.Pokemon at 0x7f9761346cd0>
```

Because our Python custom class is empty, it simply returns the address where the object is stored. In object-oriented programming, the properties of a custom object are defined by attributes, while its methods define its behavior.

In Python, the variable name *self* is used as a keyword to represent an instance of a class. It works as a handle to access the class members, such as attributes from the class methods. It is the first argument to a function called the constructor or the __init__() method and is called automatically to initialize the class

attributes with the values defined by the user. Let us look at an example:

```
class Pokemon:

    def __init__(self):
        print(“calling constructor...")

>>> poke = Pokemon()
```

INSTANCE AND CLASS ATTRIBUTES

Let's update the Pokemon class with an __init()__ constructor method that creates two attributes: name and attack. These attributes are called instance attributes.

```
class Pokemon:
    def __init__(self, name, attack):
        self.name = name
        self.attack = attack

>>> poke = Pokemon(‘charizard’,‘blast_burn’)

>>> poke.name, poke.attack
('charizard', 'blast_burn')
```

Instance methods are functions defined inside a class and can only be called from an instance of that class. Like __init__(), the first parameter of an instance method is always self.

Let's define some instance methods for our Python custom class Pokemon.

```
class Pokemon:
    def __init__(self, name, attack):
        self.name = name
        self.attack = attack

    def description(self):
        return f"{self.name} favorite attack
                          is {self.attack}"

    def speak(self, sound):
        return f"{self.name} says {sound}”

>>> pchu = Pokemon("Pikachu", “ElectroBall")

>>> pchu.description()
"Pikachu favorite attack is ElectroBall"

>>> pchu.speak("pikachu pikachu")
'Pikachu says pikachu pikachu'
```

CLASS ATTRIBUTES

Class attributes are the variables defined directly in the class that are shared by all objects of the class. Instance attributes are attributes or properties attached to an instance of a class. Instance attributes are usually defined in the constructor.Class Attributes are defined directly inside a class, and shared across all objects of the class. Class attributes are accessed using class name as well as using object with dot notation, for example using syntax *classname.class_attribute or*

object.class_attribute. Changing the value of a class attribute by using *classname.class_attribute* = *value* will be reflected to all the objects. Changing value of instance attribute will not be reflected to any other objects.

Class attributes are created by assignment statements in the suite of a `class` statement, outside of any method definition. In the broader developer community, class attributes may also be called class variables or static variables. The following class statement creates a collection of three class attributes for the `Pokemon` class.

```
class Pokemon():
    attack = 12
    defense = 10
    health = 15

    def __init__(self, name, level = 5):
        self.name = name
        self.level = level
```

Be aware of the difference between class attributes and instance attributes. In our example above, attack, defense, and health are class attributes, where name and level are instance attributes.

INHERITANCE

We extend the Pokemon class to include attributes for leveling and evolving characteristics. We use inheritance to define a new class Grass_Pokemon as a subclass that inherits attributes and methods from Pokemon, but changes some aspects. In our example, the boost values are different. For the subclass Grass_Pokemon, we show how to add an additional method, called action, which returns the string "[name of pokemon] knows a lot of different moves!".

.

```
class Pokemon():
    attack = 12
    defense = 10
    health = 15

    def __init__(self, name, level = 5):
        self.name = name
        self.level = level

    def train(self):
        self.update()
        self.attack_up()
        self.defense_up()
        self.health_up()
        self.level = self.level + 1
        if self.level%self.evolve == 0:
            return self.level, "Evolved!"
        else:
            return self.level

    def attack_up(self):
        self.attack = self.attack +
                          self.attack_boost
        return self.attack
```

```
    def defense_up(self):
        self.defense = self.defense +
                            self.defense_boost
        return self.defense

    def health_up(self):
        self.health = self.health +
                            self.health_boost
        return self.health

    def update(self):
        self.health_boost = 5
        self.attack_boost = 3
        self.defense_boost = 2
        self.evolve = 10

    def __str__(self):
        self.update()
        return "Pokemon name: {}, Level:
          {}".format(self.name, self.level)

class Grass_Pokemon(Pokemon):
    attack = 15
    defense = 14
    health = 12

    def update(self):
        self.health_boost = 6
        self.attack_boost = 2
        self.defense_boost = 3
        self.evolve = 12

    def action(self):
        return "{} knows a lot of different
          moves!".format(self.name)

>>> p0 = Pokemon("Abe")
>>> p1 = Grass_Pokemon("Bella")

>>> print(p0)
Pokemon name: Abe Level: 5
```

```
>>> print(p1)
Pokemon name: Bella Level: 5

>> p1.train()
6

>>> p0.train()
6

>>> p1.attack
17

>>> p0.attack
15

>> p2 = Pokemon("Carlos",9)

>>> p2.train()
(10, 'Evolved!')
```

METHODS FOR COMPARING OBJECTS

We now write a function method for the Pokemon class that compares two health values, and returns whether the first object's health is strictly greater than the second, as follows:

```
>>> healthier(p0,p1)
True
```

This method is slightly more complicated because it operates on two `Pokemon` objects, not just one. There are two approaches. We write an auxiliary function outside the class, which

calls a class attribute as a method bound to the first argument.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class, as you could equivalently, assign a function object to a class variable.

```
class Pokemon():


…
    # function is class attribute
    def healthier (self, poke2):
         if self.health > poke2.health
         return True
         else: return False

# auxiliary function, not a class attribute

def healthier(poke1,poke2):
    return poke1.healthier(poke2)


>>> p0.healthier(p1)
True

>>> p1.healthier(p0)
False

>>> healthier(p0,p1)
True
```

AN OOP IMPLEMENTATION OF TREE ADT

In Chapter 6 we presented a functional definition of a tree data structure. The definition was recursive since a tree was constructed with a pair consisting of root data and a list of branches, where each branch is itself a tree. Here we present an OOP design of a tree data structure.

You will see a new type of method called a repr method. The __repr__ method defines behavior which returns the string representation of an object. Typically, the __repr__ method is defined to return a string that can be executed by the interpreter and yield the same value as the given object. In other words, if you pass the returned string from the *object_name.__repr__()* method to the *eval()* function, then you will get back the same value as the value of the object. Let us take a look at an example.

```
class Tree:
    def __init__(self, label, branches=()):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = branches

     def __repr__(self):
        if self.branches:
            return 'Tree({0},
               {1})'.format(self.label,
                    repr(self.branches))
        else:
```

```
            return
     'Tree({0})'.format(repr(self.label))

     def is_leaf(self):
        return not self.branches

# Compare the following definition to the
# functional style definition for the same
# abstract object given in Chapter 6.

def fib_tree(n):
    if n == 1:
        return Tree(0)
    elif n == 2:
        return Tree(1)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        return Tree(left.label +
               right.label, (left, right))

>>> fib_tree(4)
Tree(2, (Tree(1), Tree(1, (Tree(0),
Tree(1)))))

>>> t = fib_tree(4)

>>> repr(t)
'Tree(2,(Tree(1), Tree(1,(Tree(0),
Tree(1)))))'

# repr returns a legal python expression and
# so it can be evaluated using eval() to
# construct a copy of the object

>>> new_t = eval(repr(t))

>>> new_t
Tree(2,(Tree(1), Tree(1,(Tree(0),
Tree(1)))))
```

LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 8: MUTABLE OBJECTS AND MAGIC METHODS

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

8. MUTABLE OBJECTS AND MAGIC METHODS

BY FRED ANNEXSTEIN

CHAPTER 9 OBJECTIVES

By the end of this module, you will be able to...

1. Understand programming with objects with mutable data.
2. Understand operator overloading and pythons magic methods, including __repr__, __add__, and __call__ methods.
3. Apply python programming to the shallow and deep modification of mutable objects.

LINKED LIST CLASS

Recall that a linked list, introduced in chapter 6, is composed of a first element and the rest of the list. The rest of a linked list is itself a linked list — a recursive definition. The empty list is a special case of a linked list that has no first element or rest. A linked list is a sequence: it has a finite length and supports element selection by index.

We can now implement a class with the same structure and behavior. In this OOP version, we will define its behavior using special method names that allow our class to work with the built-in `len` function and element selection

operator (square brackets or `operator.getitem`) in Python. These built-in functions invoke special method names of a class: length is computed by `__len__` and element selection is computed by `__getitem__`. The empty linked list is represented by an empty tuple, which has length 0. Here is a class definition to be used to create linked list objects.

```
class Link:
   empty = ()

   def __init__(self, first, rest=empty):
      self.first = first
      self.rest = rest

   def __getitem__(self, i):
      if i == 0:
          return self.first
      else:
          return self.rest[i-1]

   def __len__(self):
      return 1 + len(self.rest)

>>> s = Link(30, Link(40, Link(50)))
>>> len(s)
3
>>> s[1]
40
```

The definitions of __len__ and __getitem__ are recursive. The built-in Python function len invokes a method called __len__ when applied to a user-defined object argument. Likewise, the element selection operator

invokes a method called __getitem__. Thus, bodies of these two methods will call themselves indirectly. For __len__, the base case is reached when self.rest evaluates to the empty tuple, Link.empty, which has a length of 0.

The built-in isinstance function returns whether the first argument has a type that is or inherits from the second argument. isinstance(rest, Link) is true if rest is a Link instance or an instance of some sub-class of Link.

```
>>> isinstance(s,Link)
True
```

The Link class has a closure property. Just as an element of a list can itself be a list, a Link can contain a Link as its first element.

```
>>> s_1 = Link(s, Link(60))
>>> s_1
Link(Link(30, Link(40, Link(50))), Link(60))
```

The s_1 linked list has only two elements, but its first element is a linked list with three elements.

```
>>> len(s_1)
2
>>> len(s_1[0])
3
>>> s_1[0][2]
50
```

REPRESENTING CLASSES WITH STRINGS

Our implementation is basically complete, but an instance of the Link class is currently difficult to inspect. To help with debugging, we can also define a function to convert a Link to a string expression.

In Python, there are a few methods that you can implement in your class definition to customize how built-in functions that return representations of your class behave.

There are two string method traditionally used. One is called “str” and another called “repr”. The difference is that “repr” generally returns a string that can be processed by the interpreter to re-produce an exact copy of the object.

```
def __str__(self):
   if self.rest == Link.empty:
     str = ''
   else:
     str=" --> " + Link.__str__(self.rest)

   return '{0}{1}'.format(self.first, str)

def __repr__(self):
    if self.rest == Link.empty:
      repstr = ''
    else:
      repstr=", " + Link.__repr__(self.rest)

    return ‘Link({0}{1})'.
               format(self.first, repstr)
```

```
>>> s = Link(30, Link(40, Link(50)))

>>> str(s)
‘30 --> 40 --> 50'

>>> repr(s)
'Link(30, Link(40, Link(50)))’

>>> s
Link(30, Link(40, Link(50)))
```

So we see that when the interpreter reads an object, it calls the repr function on that object and returns the repr string without quotes. The major difference between using str() and repr() is found in the intended audience. repr() is intended to produce output that is mostly machine-readable (in many cases, it could be valid Python code), whereas str() is intended to be human-readable.

MAGIC METHODS

Magic methods are an important aspect of object-oriented programming in Python. Magic methods are special methods that you can define to add substantially to the usability of your classes.

These methods are sometimes called ‘dunder’ methods, since they are always surrounded by double underscores (e.g., __init__ , __len__, or __add__).

A basic magic method is the init method, `__init__`. This is the method used to define the initialization behavior of an object. The `__init__` method is not the first to get called when a new object is defined. First a method called `__new__`, which actually creates the instance, and then passes any arguments at creation on to the initializer. At the other end of the object's lifespan, there is a deletion method `__del__`.

Most collection classes will define a magic method for length, such as we have seen. We can use the length function to implement a comparison operation, in fact we are overloading the operator >.

```
def __gt__(self, other):
    return len(self) > len(other)


>>> s = Link(3, Link(4, Link(5)))
>>> t= Link(100, Link(200))
>>> s > t
True
>>> t > s
False
```

The call magic method enables an object to become function. Here we show a call magic method that enables linked list object may to call itself and apply a function to each of its elements. An alternative to this would be to

define an auxiliary function that defined a map function applied to a linked list object.

```
def __call__(self,f):
    if self == Link.empty:
      return self

    if self.rest == Link.empty:
      self.first = f(self.first)
      return self

    self.first = f(self.first)
    self.rest.__call__(f)
    return self

>>> s = Link(30, Link(40, Link(50)))
>>> s(lambda x:x*x)
Link(900, Link(1600, Link(2500)))
```

OPERATOR OVERLOADING

Usability is enhanced if user can apply commonly used arithmetic and comparison based operators. Python has many magic methods designed to support intuitive comparisons between objects using symbolic operators. They also provide a way to override the default Python behavior for comparisons of objects. Here is an example of overloading the

+ operator, so that two linked lists can be added together.

```
def __add__(self, other):
    if self is Link.empty:
       return other
    else:
       return Link(self.first,
          Link.__add__(self.rest, other))


>>> s = Link(3, Link(4, Link(5)))
>>> t = Link(100, Link(200))

>>> s + t
Link(3, Link(4, Link(5, Link(100,
Link(200)))))

>>> t + s
Link(100, Link(200, Link(3, Link(4,
Link(5)))))
```

Here's the list of those common comparison methods and what they do:

```
__eq__(self, other)
Defines behavior for the equality
operator, ==.

__ne__(self, other)
Defines behavior for the inequality
operator, !=.

__lt__(self, other)
Defines behavior for the less-than
operator, <.

__gt__(self, other)
```

```
Defines behavior for the greater-than
operator, >.

__le__(self, other)
Defines behavior for the less-than-or-equal-
to operator, <=.

__ge__(self, other)
Defines behavior for the greater-than-or-
equal-to operator, >=.
```

Here's the list of those common binary arithmetic operators that can implement magic methods:

```
__add__(self, other)
Implements addition.

__sub__(self, other)
Implements subtraction.

__mul__(self, other)
Implements multiplication.

__floordiv__(self, other)
Implements integer division using
the // operator.

__div__(self, other)
Implements division using the / operator.
```

Nearly all common arithmetic binary and unary operators can be converted to a magic method for a class. For more details see https://rszalski.github.io/magicmethods/#comparisons

LINK LIST CLASS WITH MAGIC METHODS

Here is a summary of all the code that we developed for this class.

```
class Link:
   empty = ()

   def __init__(self, first, rest=empty):
      self.first = first
      self.rest = rest

   def __getitem__(self, i):
      if i == 0:
          return self.first
      else:
          return self.rest[i-1
                           ]
   def __len__(self):
      return 1 + len(self.rest)

   def __gt__(self, other):
       return len(self)> len(other)

   def __str__(self):
       if self.rest == Link.empty:
         str = ''
       else:
         str=" --> " + Link.__str__(self.rest)
       return '{0}{1}'.format(self.first, str)

   def __repr__(self):
      if self.rest == Link.empty:
        repstr = ''
      else:
        repstr=", " + Link.__repr__(self.rest)
      return 'Link({0}{1})'.format(self.first,
                                   repstr)

   def __add__(self, other):
      if self is Link.empty:
        return other
      else:
        return Link(self.first, Link.__add__(
                                self.rest, other))
```

A TREE CLASS WITH REPR

Trees can also be represented by instances of user-defined classes, rather than nested instances of built-in sequence types. Recall that a tree is a data structure that has as an data attribute and an attribute that is a sequence of branches that are also trees.

Previously, we defined trees in a functional style. Here is an alternative class definition, which we will use to illustrate new features that are magic methods.

We define the trees so that they have internal values at the roots of each subtree. The internal value is called a label in the tree. The Tree class below represents such trees, in which each tree has a sequence of branches that are also trees. A repr function is defined to return a string that could be evaluated to a Tree.

```
class Tree:
   def __init__(self, label, branches=()):
       self.label = label
       for branch in branches:
           self.branches = branches

   def is_leaf(self):
            return not self.branches

   def __repr__(self):
       if self.branches:
           return 'Tree({0},
               {1})'.format(self.label,
               repr(self.branches))
       else:
```

```
            return
                'Tree({0})'.format(repr(self.
                label))
```

The Tree class can represent, for instance, the values computed in an expression tree for the recursive implementation of fib, the function for computing Fibonacci numbers. The function fib_tree(n) below returns a Tree that has the nth Fibonacci number as its label and a trace of all previously computed Fibonacci numbers within its branches.

```
def fib_tree(n):
        if n == 1:
            return Tree(0)
        elif n == 2:
            return Tree(1)
        else:
            left = fib_tree(n-2)
            right = fib_tree(n-1)
            return Tree(left.label +
right.label, (left, right))

>>> fib_tree(5)
Tree(3, (Tree(1, (Tree(0), Tree(1))),
Tree(2, (Tree(1), Tree(1, (Tree(0),
Tree(1)))))))
```

INTRODUCING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 9: MACHINE LEARNING AND DECISION TREES

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

9. MACHINE LEARNING AND DECISION TREES

BY FRED ANNEXSTEIN

CHAPTER 9 OVERVIEW

In this module we will be covering an introduction to machine learning libraries using python. We focus on the popular scikit-learn or sklearn library and demonstrate several examples of solving classification problems using sklearn's version of decision trees and random tree methods.

CHAPTER 9 OBJECTIVES

By the end of this chapter, you will be able to...

Understand some typical machine learning problems and solution approaches.

Understand the numpy library and ndarray type.

Run a machine learning library using several estimators or models.

Understand the interfaces of the sklearn library.

Recognize and describe the accuracy of machine learning models.

Understand the classification models known as Decision Trees and Random Forests.

Practice hyper-parameter tuning in the context of a lab for random forest tuning.

MACHINE LEARNING PROBLEMS

A machine learning problem typically considers a set of n samples of data, and then tries to predict properties of unknown data. Usually each sample is a simple list of numbers. Each number is the list is associated with a particular attribute or a feature of the problem.

Machine learning problems generally fall into the basic categories of either supervised learning or unsupervised learning. Classification problems are considered supervised learning since they are based on having data that have labeled classes identified. Density estimation and clustering are examples of unsupervised learning and do not depend on having labeled attributes.

When using Python for solving machine learning problems it is typical to store data in what is known as a Numpy array or an ndarray. This data type is popular for doing high-performance data manipulation required by machine learning algorithms. Ndarrys are useful within the context of general python programming, as an alternative to slow list operations.

NUMPY NDARRAY TYPE

We will be using the data type known as ndarrays in this chapter, rather than another sequence type such as a list. In general, in most programming languages, arrays are stored contiguously in memory and are used to efficiently store data of one fixed type of fixed size memory blocks.

The Numpy library defines a class ndarray for creating array objects. Each ndarray objects is used to represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.) Elements in Numpy arrays are accessed by using square brackets and can be initialized by using nested Python Lists.

For machine learning applications we will use arrays to store table of attribute values (usually numbers), all of the same type, and indexed by a tuple of positive integers. The number of dimensions of the array is called the rank of the array. A tuple of integers giving the size of the array along each dimension is known as shape of the array.

Here is a sample script illustrating the

attributes of 1D and 2D arrays using numpy.ndarray type.

```
import numpy as np

# Creating 1D array object
arr1 = np.array( [ 1.0, 2.0, 3.0, 4.0] )

# Creating array object
arr2 = np.array( [[ 1, 2, 3],
                  [ 4, 2, 5]] )

# Printing type of arr object
print("Array is of type: ", type(arr1))

# Printing array dimensions (axes)
print("No. of dimensions arr1: ", arr1.ndim)
print("No. of dimensions arr2: ", arr2.ndim)

# Printing shape of array
print("Shape of array arr1: ", arr1.shape)
print("Shape of array arr2: ", arr2.shape)

# Printing size (total number of elements)
of array
print("Size of array: ", arr1.size)
print("Size of array: ", arr2.size)

# Printing type of elements in array
print("Array stores elements of type: ",
arr1.dtype)
print("Array stores elements of type: ",
arr2.dtype)


Output:
Array is of type:  <class 'numpy.ndarray'>
No. of dimensions arr1:  1
No. of dimensions arr2:  2
```

```
Shape of array arr1:  (4,)
Shape of array arr2:  (2, 3)
Size of array:  4
Size of array:  6
Array stores elements of type:  float64
Array stores elements of type:  int64
```

SUPERVISED VS UNSUPERVISED LEARNING

Supervised learning problems have as input a data table with an additional attribute that we want to be able to predict. Supervised machine learning falls into two categories—classification and regression. Solutions to classification problems have a set of discrete possible outputs, which is a prediction of the class any new sample data belongs to.

In solutions to regression problems the desired output consists of one or more continuous variables. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight. Another example would be predicting local temperatures, given a weather time series.

In contrast, for unsupervised learning problems the training data consists of a set samples without any corresponding target values. The goal in such problems may be to discover groups of similar examples within

the data - this is the problem of data clustering. Another example is to determine the distribution of data within the input space — this is the problem of density estimation. Finally, there is the example of transforming the data from a high-dimensional space down to a small number (two or three dimensions) for the purpose of analysis or visualization - this is the problem of dimensionality reduction.

CLASSIFICATION

In a classification problem, samples belong to two or more target classes, and the goal is to learn a model from the already labeled data, which can be used to predict the class of any unlabeled data.Classification is a discrete form of supervised learning where one has a limited number of categories for each of the say n samples provided.

Binary classification uses two classes, such as “true” of “false” for boolean outputs. For example, we may want to learn to label input email messages as “spam” or “not spam” in an email classification application.

Multi-classification uses more than two classes. For example, in the problem of digit image recognition we may have 10 classes

of digit possibilities 0 through 9, which the case in a digit-image dataset.

A multi-classification scheme looking at movie descriptions might try to classify them by genre, such as “action,” “adventure,” “fantasy,” “romance,” “history” and so forth.

TRAINING SET AND TESTING SET

Machine learning is concerned with learning a model to representing some patterns or properties of a data set, and then testing those properties against the model using another data set. A common practice is to evaluate an algorithm by splitting a given learning data set into two. We call one of those sets the training set, on which we set to learn some properties. The other set we call the testing set, on which we run test to see if the model accurately predicts the properties, hence producing an accuracy score. The next examples should help clarify the use of these sets.

SCIKIT-LEARN

Scikit-learn or simply sklearn has many python library of modules for many machine learning tasks. In this chapter we will begin to explore the library’s features for various

classification, regression and clustering algorithms.

The sklearn modules are designed to interoperate with standard numerical and scientific python libraries, specifically numpy and scipy. Sklearn is largely written in python, and uses numpy extensively for high-performance linear algebra and array operations.

If you need to install scikit-learn, you can do so using the following command.

```
$ conda install -c anaconda scikit-learn
```

In this introductory chapter we will demonstrate learning experiments with sci-kit learn's tree-based models known as *decision tree classifiers*. This classifier is ready to use right out of the box when using this sklearn package.

SKLEARN DATASETS

The sklearn software packages comes with a few small standard datasets that do not require you to download files from some external website. These datasets come with a small number of samples with a limited number of features, which makes it easy to experiment with. Using these data sets will

help you to understand how sklearn expects data to be named and shaped. The data sets can also be found at the following link: https://scikit-learn.org/stable/datasets/toy_dataset.html

At the end of the chapter will be links to information on loading external data sets when using sklearn.

LOADING AN EXAMPLE DATASET FROM SKLEARN

In this example we show how to load the classic dataset with sampled data about Iris flowers containing flower names and measurements of the organs of mature flowers. This data set will show us how a solution to a basic classification problem is solved using sklearn. We first input the data for our program.

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
```

The call of the function `datasets.load_iris()` creates and returns a dictionary like object called a Bunch object, as we can see from using the type() command.

```
>>> type(iris)
sklearn.utils.Bunch
```

Within the Bunch is the data (measurements), the target values (referencing flower names) and various metadata, as we can see using the dir() command.

```
>>> dir(iris)
['DESCR',
 'data',
 'data_module',
 'feature_names',
 'filename',
 'frame',
 'target',
 ‘target_names']
```

The `.data` member of the iris Bunch is simply a 2D numpy ndarray with shape pair determined by the number of samples, and the number of attributes.

```
>>> type(iris.data)
numpy.ndarray
>>> iris.data.shape
(150, 4)
```

So this iris data set has 150 samples each with 4 features each representing a measurement of a different part of the sample flower. The first flower sample in the data has

four measurements [5.1, 3.5, 1.4, 0.2], as we see below.

```
>>> iris.data
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       …
```

Each sample also has a labeled target of one of three possible class values [0,1,2], representing three species of irises, and stored in a numpy ndarray referenced by the `.target` member of iris Bunch.

```
>> iris.target
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2])
-
```

The values in the target array are just indexes for the class of each flower. The names of each flower species is stored in an array of three strings called *.target_names.*

```
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'],
dtype='<U10')
```

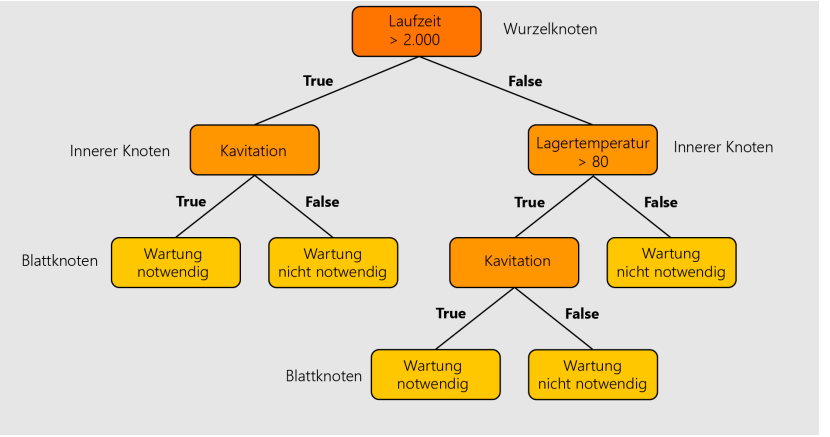
The dtype attribute at the end dtype='<U10' of the array indicates that the data type held by the numpy ndarray is unicode and this gives information about how Numpy stores the array.

CLASSIFICATION VIA DECISION TREES

Decision trees are labeled binary trees (See Chapter 6) and are very useful for constructing a supervised learning model. Decision trees are often used in classification and regression problems. The goal of a decision tree is to create a model that can accurately predict the value of a target variable by applying easy decision rules. These rules are inferred from the data during the learning process. A tree can be seen as a piecewise constant approximation, that is as the value of an attribute crosses a threshold the classification can change.

Decision trees contain at each non-leaf node an if-then-else decision rule followed by new branches of the tree. The deeper the tree, the more complex the decision rules become, and as a result deep-trees may exhibit over-fitting, which is considered a problem of poor generalization from the data.

The basic structure of a decision tree is shown in the figure below, where the yellow leaf nodes label the class, and the orange internal nodes are labeled with T/F decision test.



DECISION TREE CLASSIFIERS IN SKLEARN

To use decision trees within sklearn we use the tree module method called DecisionTreeClassifier, which takes as input two arrays: an array X, of shape `(n_samples, n_features)` holding the training samples, and an array Y of integer values, with shape `(n_samples)`, holding the class labels for the training samples

A DecisionTreeClassifier is capable of both binary classification — where the labels are [-1, 1]), and multiclass classification — where the labels are [0, …, K-1]. In our example using the iris data we have target multiclass classification with labels [0,1,2]

TRAINING AND TESTING A DECISION TREE CLASSIFIER

We will use the standard sklearn function called train_test_split() to split up our data set to create training and testing data sets.

We then call the method tree.DecisionTreeClassifier() to construct a decision tree object named **`clf`** (a standard name for classifier in sklearn). We then fit the data by calling the fit function. And finally we test the accuracy of the model on both the training data and the testing data. Here is the script.

```
from sklearn.model_selection import
train_test_split
from sklearn.datasets import load_iris
from sklearn import tree

iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test =
train_test_split(X, y, random_state=0)

clf = tree.DecisionTreeClassifier()
clf.fit(X_train, y_train)
```

Once we fit the model we can test for accuracy both on the training data and on the testing data. In this example, we find the model is 100% accurate on training data, which is not surprising, since the model saw all the labels of the training data. We find that the model is 97.3% accurate on testing data, which is a pretty solid result.

```
>>> clf.score(X_train, Y_train))
1.0
>>> clf.score(X_test, Y_test))
0.9736842105263158
```

SKLEARN.TREE.DECISIONTREECLASSIFIER

The decision tree classifier has an attribute called tree_ which will allows access to low level attributes such as node_count, the total number of nodes, and max_depth, the maximal depth of the tree. It also stores the entire binary tree structure, represented as a number of parallel arrays. Each node of the tree has a unique index value spanning the arrays, that is the ith element of each array holds information about the node `i`. The standard convention is that node 0 is the tree's root.

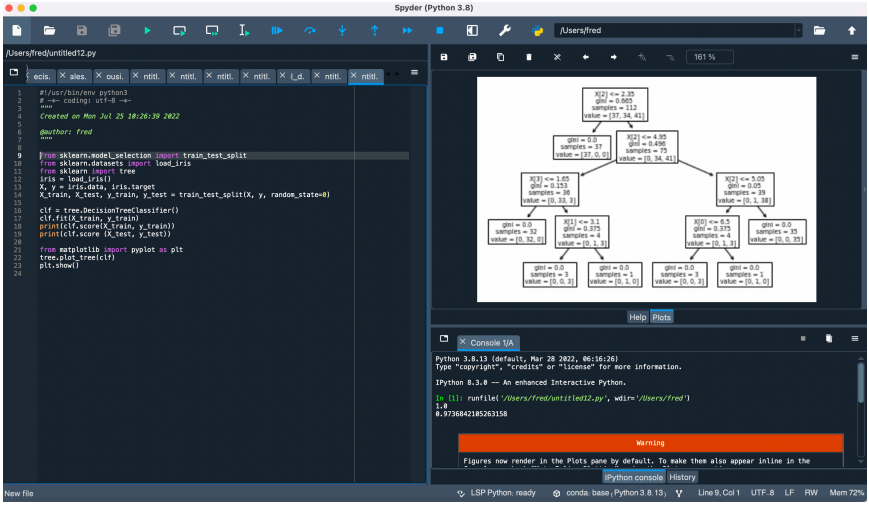
Some of the information arrays only apply to either leaves or internal split nodes. In this case the array values of certain indexes may not be relevant and should be ignored. For example, the information arrays `feature` and `threshold` only apply to split nodes. The values for leaf nodes in these arrays are therefore ignored.

Here are some of these tree information arrays stored in a sklearn decision tree,

`children_left[i]`: id of the left child of node `i` or -1 if leaf node
`children_right[i]`: id of the right child of node `i` or -1 if leaf node
`feature[i]`: feature used for splitting node `i`
`threshold[i]`: threshold value at node `i`
`n_node_samples[i]`: the number of training samples reaching node `i`
`impurity[i]`: the impurity at node

VISUALIZATION OF THE DECISION TREE

We can visualize the resulting sklearn decision tree to see each of the nodes and the logic of each split. We do this by referencing matplotlib and adding three lines of code at the end of the above sample



script:

```
from matplotlib import pyplot as plt
tree.plot_tree(clf)
plt.show()
```

Visualization of Decision tree for Iris dataset.

GINI SCORING

In the image generated above you will see that each node has a gini score associated with it. This score is a statistically measured value between 0 and 1 and used by the algorithm to determine the next decision variable to pick for the branching question.

The gini score is a measure of statistical dispersion, and is intended to represent the inequality between the classes. For

decision trees, more inequality is usually better since we are using information to determine the particular target. The gini score was initially developed by the statistician and sociologist Corrado Gini and used to measure income inequality.

For decision trees, lower gini scores are better. A gini score of 0 expresses no impurity or perfect targeting, where all the samples remaining are of the same class. While a gini score of 1 (or 100%) expresses maximal impurity where all classes are equally represented.

Consider a dataset D that contains samples from k=3 classes. The probability or equivalently the fraction of samples belonging to class i at a given node can be denoted as p[i] and the gini score of D is defined as 1 minus the sum of the squares of these probabilities. Here is the gina-score function in code.

```
gini-score = lambda p,k:
       1 - sum([ p[i]**2 for i in range(k)])
```

We can allow our tree to be determined or generated by those branching questions which minimize the gini scores.

FROM DECISION TREES TO RANDOM FORESTS

One way of improving the results of the decision tree model for large and more complex data sets is to use more than one decision tree and take a weighted average of the results.

This is the goal of general machine learning strategy called *ensemble methods***.** The idea here is to combine the predictions of several base estimators. Two families of ensemble methods are usually distinguished. In *averaging methods*, the driving principle is to build several estimators independently and then to average their predictions. The combined estimator is usually better than any of the single base estimators. Statistics often show that by taking averages, the variance of the output is usually reduced.

By contrast, in *boosting methods* the base estimators are built sequentially and one tries to reduce the bias of the combined estimator in each stage. The motivation is to combine several weak models to produce a more powerful ensemble.

The sklearn.ensemble module includes an averaging algorithm based on randomized *decision trees* called the RandomForest algorithm. The prediction of the ensemble is given as the averaged integer prediction of individual decision tree classifiers.

As with the other classifiers, forest classifiers have to be fitted with two arrays: an array X of shape `(n_samples, n_features)` holding the training samples, and an array Y of shape `(n_samples)` holding the target values.

In random forests each tree in the ensemble is built from a sample drawn with replacement from the training set. Furthermore, when splitting each node during the construction of each tree, the best split is found either from all input features or a random subset of size `max_features`.

The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit high variance and tend to overfit producing rules too closely matching the data. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant, and thus yields an overall better model.

In contrast to the original algorithm designers, the sklearn implementation combines classifiers by averaging their probabilistic prediction, instead of letting each classifier vote for a single class.

READING DATA FILES WITH PANDAS

We will be using a library called Pandas to help us read in a data file into our programs. You may need to install pandas with the following command line:

```
$ conda install pandas
```

With pandas we can read in a .csv datafile as follows:

```
data = pd.read_csv('/Users/fred/Desktop/
                                    datafile.csv')
```

We will be covering more features of the important Pandas library in the following chapters.

LAB #9

HEART DISEASE CLASSIFICATION

For this lab you will run computer experiments to find good classifiers for the standard heart disease data set. Please download this dataset to your Desktop from Canvas.

Here is starter code, which you can read in the data and experiment with it. Please try to compile and run this code first. For this lab you will take this starter code and add code to carry out a set of experiments.

```
import pandas as pd
```

```
import numpy as np

heart_disease = pd.read_csv('/Users/fred/
Desktop/heart.csv')

X = heart_disease.drop(['target'] , axis=1)
Y = heart_disease['target']

from sklearn.ensemble import
RandomForestClassifier

clf =
RandomForestClassifier(n_estimators=10)

from sklearn.model_selection import
train_test_split

X_train, X_test, Y_train, Y_test =
train_test_split(X, Y, test_size = 0.2)

clf.fit(X_train, Y_train)
y_pred = clf.predict(X_test)


print(clf.score(X_train, Y_train))
print(clf.score(X_test, Y_test))
```

It is recommended that you follow this video tutorial on using sklearn. For the lab you should view the video between the 5:00 minute and 19:00 minute marks. The video will explain all the steps used in the starter code above, as well as showing you how to run an experiment with manual hyper-parameter tuning.

https://www.youtube.com/watch?v=BXkqEXjBf5s&ab_channel=MachineLearni

ng

For the lab you will vary the number of default estimators for this classifier, as shown in the video. This tuning parameter is already named n_estimators, and this parameter specifies the number of random trees used by the ensemble classifier.

You will write code to search to find the number that gives the best accuracy on the test data, X_test. Report the scores for your results given by clf.score(X_test, Y_test) over a wide range of n_estimator values.

For the lab you should summarize your results in a concise data table. Finally, put this data table as a comment at the top of your code submission.

OPTIONAL ADDITIONAL PROJECT

Use the classic Titanic dataset to fit a binary classification decision tree model using sklearn's DecisionTree and Random Forest Classifiers. What is the accuracy you are able to achieve.

INTRODUCING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 10: PANDAS AND MATPLOTLIB

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

10. PANDAS AND MATPLOTLIB

BY FRED ANNEXSTEIN, PHD

CHAPTER 10 OVERVIEW

In this module we will be programming in a style known as declarative programming, where we abstract away the control flow for the logic required for the software to perform an action. In declarative programming we use statements and commands for determining what the task or desired outcome is. In this module we will be using a collection of popular python libraries for working with table data including, Numpy, Pandas, and Matplotlib.

CHAPTER 10 OBJECTIVES

By the end of this chapter, you will be able to...

1. Understand the syntax and usage of a declarative programming style using Python.
2. Understand how to work with Matplotlib library for basic plotting, including line plots, bar plots, and scatterplots. More advanced plotting such as trig functions and fractals.
3. Understand how to work with Pandas series and data frames.
4. Understand how to create and inspect a data frame object.

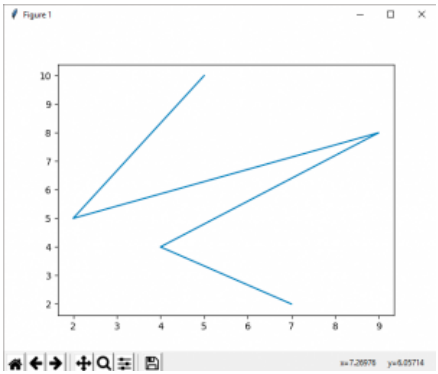
5. Understand how to aggregate and visualize statistical data with Pandas and Matplotlib.
6. Understand how to generate a scatterplot of two normalized variables and analyze their correlation.

BASIC PLOTS IN MATPLOTLIB

Matplotlib comes with a wide variety of plots. Plots helps people to understand trends, patterns, and to investigate correlations. Plots are used typically as instruments for reasoning about quantitative information. Some example code for matplotlib plots are covered here, including line plots, bar plots, and scatterplots. We start with two lists in each case, call a command to build the plot, and finally call a command to show the plot.

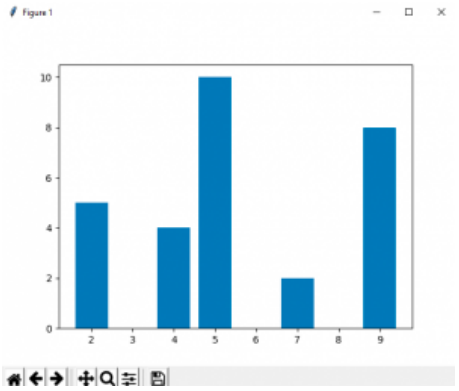
LINE PLOT

```
from matplotlib import
pyplot as plt

x = [5, 2, 9, 4, 7]
y = [10, 5, 8, 4, 2]

plt.plot(x,y)
plt.show()
```



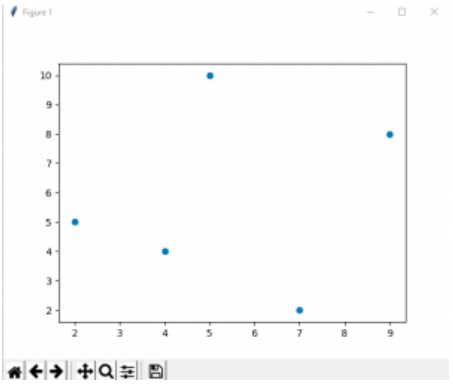
BAR PLOT

```
from matplotlib
import pyplot as
plt

x = [5, 2, 9, 4, 7]
y = [10, 5, 8, 4,
2]

plt.bar(x,y)
plt.show()
```



SCATTERPLOTS

```
from matplotlib import
pyplot as plt

x = [5, 2, 9, 4, 7]
y = [10, 5, 8, 4, 2]


plt.scatter(x, y)
plt.show()
```

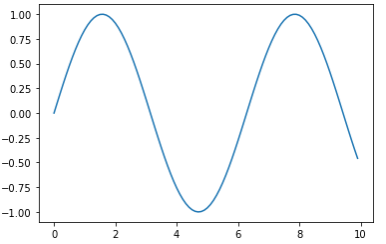


LINE PLOT FOR A TRIG FUNCTION

We can plot the curve of a trig function using Numpy arange command to obtain an array of numbers over an interval. We then use the ordinary plot to get a graph of the

sine function, for example, as follows:

```
import numpy as np
import matplotlib.pyplot as
plt

x = np.arange(0,10,.1)
y = np.sin(x)

plt.plot(x,y)
plt.show()
```

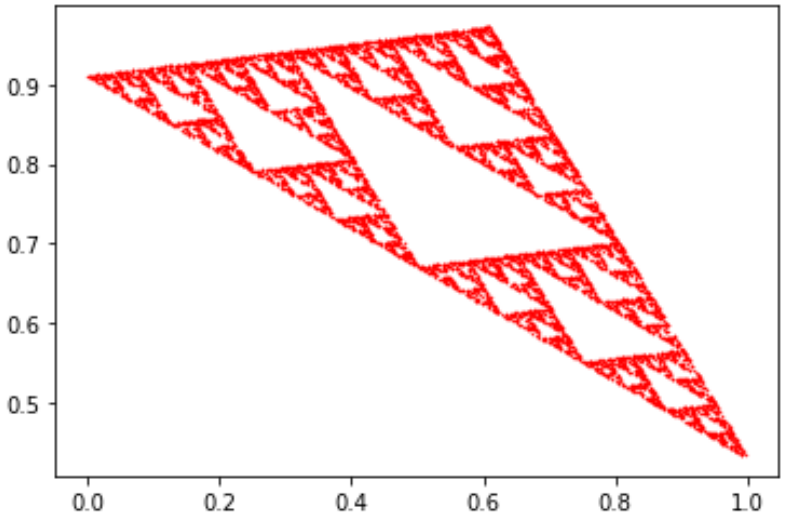


GENERATING A FRACTAL

In this next example we will use a scatterplot to visualize a fractal image known as Sierpiński triangle. The fractal is generated by choosing 3 corners of a triangle at random, as well as a random starting point. We then iteratively choose a random corner of the triangle and move to the midpoint between the current random point and the random corner. The image generated gives a recursive pattern with infinitely nested similar triangle.

```
import numpy as np
import matplotlib.pyplot as plt
from random import random, randint

# Three corners of an random triangle
corner = [(random(), random()),
(random(),random()), (random(),random())]


def midpoint(p, q):
```

```
    return (0.5*(p[0] + q[0]),
            0.5*(p[1] + q[1]))

N = 10000
x = np.zeros(N)
y = np.zeros(N)

x[0] = random()
y[0] = random()

for i in range(1, N):
    k = randint(0, 2)
    x[i], y[i] = midpoint( corner[k],
                           (x[i-1], y[i-1]) )

plt.scatter(x, y, s=0.01, c="red")
plt.show()
```



PANDAS

Pandas is an extremely popular Python library for data table creation and manipulations. Pandas makes abundant use of NumPy's ndarray, which was a data class introduced in the last chapter.

Pandas has two key data structures: the Series, which is one dimensional, and DataFrames, which is two dimensional. We will use DataFrame to provide a size-mutable, two-dimensional structure for data tables made up of three components: rows and columns (which are labeled) and data values.

PANDAS DATAFRAME

A DataFrame is an enhanced two-dimensional array. DataFrames can have custom row and column indices, and offer additional operations and capabilities that make them more convenient for many data-science oriented tasks. DataFrames also support missing data.

DataFrames let you organize tabular information, allowing you to view it more easily than nested lists. Generally speaking, in the DataFrame grid, each row corresponds with an instance or data sample, and each

column contains variable data for one attribute. The data in the columns can contain numeric, alphanumerical characters or logical data and typically are of the same type, although they do not have to be.

Each column in a DataFrame is called a series, an object of the class pandas.core.series.Series. The Series representing each column may contain different element types, however we will only be storing series of numbers.

To create a Python Pandas DataFrame, we can load existing datasets using a CSV file. However, next we show how to create a Dataframe grid directly from a dictionary using multiple lists. Each key/value pair of the dictionary will become a new column, see the example that follows:

```
import pandas as pd

temps_dict = {
    'St Thomas': [87, 96, 70],
    'Key West': [100, 87, 90],
    'San Juan': [94, 77, 90],
    'Havana': [100, 81, 82],
    'Miami': [83, 65, 85]}

temps = pd.DataFrame(temps_dict)


>>> type(temps)
pandas.core.frame.DataFrame
```

```
>>> temps
  St Thomas  Key West  San Juan  Havana  Miami
0         87       100        94     100     83
1         96        87        77      81     65
2         70        90        90      82     85

>>> type(temps.'Miami')
pandas.core.series.Series

>>> temps.Miami
0    83
1    65
2    85
Name: Miami, dtype: int64
```

CUSTOMIZING A DATAFRAME'S INDICES WITH THE INDEX ATTRIBUTE

We can use the index attribute to change the DataFrame's row indices from sequential integers to strings, which helps readability.

```
>>> temps.index=['Friday', 'Saturday', 'Sunday']

>>> temps
           St Thomas  Key West  San Juan  Havana  Miami
Friday            87       100        94     100     83
Saturday          96        87        77      81     65
Sunday            70        90        90      82     85
```

Also referred to as Subset Selection, indexing simply means using the .iloc and .loc indexers to select some or all of the DataFrame's rows or columns.

To select one column, place the column's

name between brackets. To select one row, place the row's name as in index to the .loc object in class pandas.core.indexing._LocIndexer. The code would look similar to this:

```
>>> temps['Miami']
0    83
1    65
2    85
Name: Miami, dtype: int64

>>> temps.loc['Friday']

St Thomas     87
Key West     100
San Juan      94
Havana       100
Miami         83
Name: Friday, dtype: int64
```

Use DataFrame.loc[] or pass the integer's location as an index to the iloc[] object to select multiple rows and columns. Here is what the code might look like:

```
>>> temps.loc[['Friday','Sunday']]

        St Thomas  Key West  San Juan  Havana  Miami
Friday         87       100        94     100     83
Sunday         70        90        90      82     85


# Return first 3 columns of 2 outer rows
>>> temps.iloc[[0, 2], 0:3]

        St Thomas  Key West  San Juan
Friday         87       100        94
Sunday         70        90        90
```

One of pandas' more powerful selection capabilities is Boolean indexing. For example, we can create a table selecting all the high temperatures —that is, those that are greater than or equal to 90, as follows:

```
>>> temps[temps >= 90]

          St Thomas  Key West  San Juan  Havana  Miami
Friday          NaN     100.0      94.0   100.0    NaN
Saturday       96.0       NaN       NaN     NaN    NaN
Sunday          NaN      90.0      90.0     NaN    NaN
```

We can locate an individual attribute by specifying the row and column indexes, as follows:

```
>>temps.at['Saturday', 'San Juan']
77
```

CHAPTER 10 LAB

For Lab 10 we will load data from the **Superhero Movie Dataset —** you will find this dataset at the end of the chapter. You will then perform some simple manipulations and plot the data as a means of exploring the features in this dataset. Let us create the lab program from scratch, as follows:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from pandas import DataFrame, Series
```

Next, download the dataset and save it as a .csv file. You should be able to do this using Numbers on Mac or Excel on Windows. Save the data set as 'dataset.csv'.

We will read the data from the .csv file using the Pandas pd.read_csv() method. Since this particular data set does not include a header row, we must name each column series in the file. Try to execute this code:

```
sh_raw =
  pd.read_csv(‘/Users/fred/Desktop
     dataset.csv’,  header=None,
     names=
[‘Year','Title','Comic','IMDB','RT',
‘CompositeRating','OpeningWeekendBoxOffice',
'AvgTicketPriceThatYear','EstdOpeningAttenda
nce','USPopThatYear'])

print(sh_raw.head(5))
```

When you execute it, you should see output like this:

```
 Year         Title  ... EstdOpeningAttendance  USPopThatYear
NaN  1978.0      Superman  ...          3190317.521    222584545.0
NaN  1980.0   Superman II  ...          5241830.112    227224681.0
NaN  1982.0   Swamp Thing  ...                  NaN    231664458.0
NaN  1983.0  Superman III  ...          4238843.492    233791994.0

[5 rows x 10 columns]
```

Since we will be analyzing box office numbers, we need to clean up the dataset and exclude movies where this data is missing. those columns have **Not a Number** `NaN` in the **OpeningWeekendBoxOffice** Series. We will use the Numpy function `isfinite()` to check for value numbers in this column. Try to execute the following:

```
sh = sh_raw[np.isfinite(
           sh_raw.OpeningWeekendBoxOffice)]
print(sh.head(5))
```

Output:

```
Year            Title  ... EstdOpeningAttendance  USPopThatYear
NaN  1978.0         Superman  ...           3190317.521    222584545.0
NaN  1980.0      Superman II  ...           5241830.112    227224681.0
NaN  1983.0     Superman III  ...           4238843.492    233791994.0
NaN  1984.0        Supergirl  ...           1707812.202    235824902.0
NaN  1986.0  Howard the Duck  ...           1366613.477    240132887.0
```

You should notice that Swamp Thing has been omitted from the output. That happened because we did not have the OpeningWeekendBoxOffice value for this title.

With our dataset cleaned, we now add the calculated columns required to perform our analysis.

We wish to compare Rotten Tomatoes ratings to IMDB ratings. To do this meaningfully, we have to normalize them first, since they have different scoring ranges. Normalization is done by dividing the original score by the maximum possible value, and thereby obtaining a normalized

score between 0.0 and 1.0. Once we produce two normalized series of numbers, we can visualize there correlation by plotting a scatterplot. Enter the following code and execute it.

```
# Normalize the scores

imdb_normalized = sh.IMDB / 10

sh.insert(10,'IMDBNormalized',imdb_normalize
d)

rt_normalized = sh.RT/100

sh.insert(11, 'RTNormalized', rt_normalized)
```

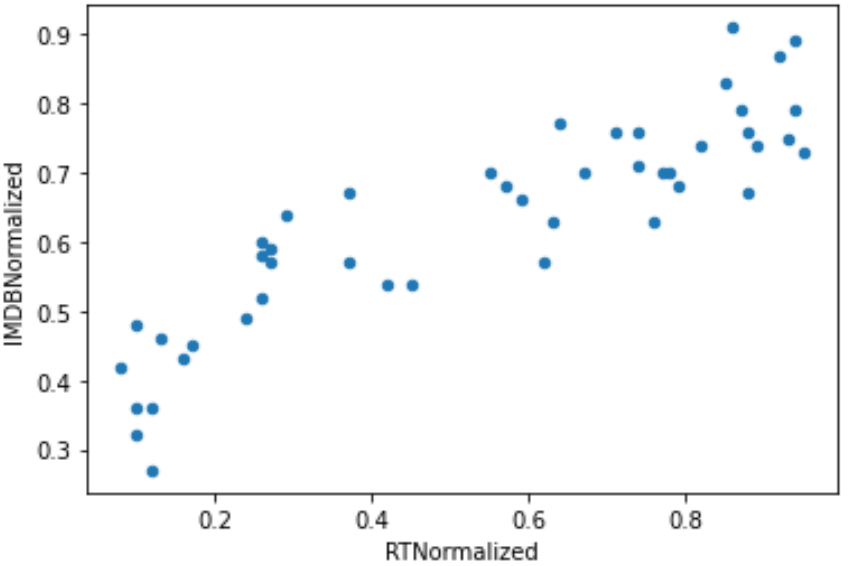
With our scores normalized, let's make our first scatter plot, and explore the relationship between Rotten Tomatoes and IMDB ratings for each movie. Try to execute the following:

```
sh.plot.scatter(x ='RTNormalized',
                y ='IMDBNormalized')
plt.show()
```

Here is the output you should see:

At a glance you can see there is a positive correlation — a trending of points up and to the right, which one should expect from two different movie ratings services.

We now want to calculate the correlation coefficient and verify how strong of a correlation this is. And, lucky for us Pandas



provides a corr() method to calculate correlations. Rather than do this to the entire DataFrame, we select the two normalized columns in question. Try to execute the following:

```
print(sh[[‘RTNormalized','IMDBNormalized']].
corr())
```

We find that the correlation is 0.88836, which, indeed, is a high positive correlation.

The Pandas describe() method makes it easy to get summary statistics for our data, including mean, standard deviation, and percentiles. Try to execute the following:

```
print(sh[['RTNormalized','IMDBNormalized']].
describe())
```

The 25th percentile is the value at which 25% of the answers lie below that value, and 75% of the answers lie above that value. From the output from the previous command it is interesting to note that in the 25th percentile for Rotten Tomatoes there are more lower ratings for the same movies than IMDB. See if you can verify that from the output.

Required Lab Questions:

There are no doctests for this lab. Please upload your modified code that includes answers to the following questions:

1. Define a command to show only `'DC'` comic movies fromthe `sh` DataFrame.
2. Define a command to show the Year, Title and OpeningWeekendBoxOffice columns from the `sh` DataFrame.
3. Define a command to show the Year and Title of only `'Marvel'` movies from the `sh` DataFrame.
4. Define a command to plot a `line()` for the AvgTicketPriceThatYear with Year on the x axis. Make the line Black.

Superhero Movie Dataset to Use for the Lab:

```
1978,Superman,DC,7.3,95,84,7465343,2.34,3190317.521,222584545
1980,Superman II,DC,6.7,88,77.5,14100523,2.69,5241830.112,227224681
1982,Swamp Thing,DC,5.3,60,56.5,,2.94,,231664458
1983,Superman III,DC,4.9,24,36.5,13352357,3.15,4238843.492,233791994
1984,Supergirl,DC,4.2,8,25,5738249,3.36,1707812.202,235824902
1986,Howard the Duck,Marvel,4.3,16,29.5,5070136,3.71,1366613.477,240132887
1987,Superman IV: The Quest for Peace,DC,3.6,10,23,5683122,3.91,1453483.887,242288918
1989,Batman,DC,7.6,71,73.5,40489746,3.97,10198928.46,246819230
1989,The Return of Swamp Thing,DC,3.9,40,39.5,,3.97,,246819230
1989,The Punisher,Marvel,5.4,24,39,,3.97,,246819230
1992,Batman Returns,DC,7,78,74,45687711,4.15,11009086.99,255029699
1995,Batman Forever,DC,5.4,42,48,52784433,4.35,12134352.41,262803276
1997,Batman & Robin,DC,3.6,12,24,42872605,4.59,9340436.819,267783607
1997,Steel,DC,2.7,12,19.5,870068,4.59,189557.2985,267783607
1998,Blade,Marvel,7,55,62.5,17073856,4.69,3640481.023,270248003
2000,X-Men,Marvel,7.4,82,78,54471475,5.39,10106025.05,282171957
2002,Blade II,Marvel,6.6,59,62.5,32528016,5.81,5598625.818,287803914
2002,Spider-Man,Marvel,7.4,89,81.5,114844116,5.81,19766629.26,287803914
2003,Daredevil,Marvel,5.4,45,49.5,40310419,6.03,6684978.275,290326418
2003,Hulk,Marvel,5.7,62,59.5,62128420,6.03,10303220.56,290326418
2003,X2,Marvel,7.6,88,82,85558731,6.03,14188844.28,290326418
2004,Blade: Trinity,Marvel,5.8,26,42,16061271,6.21,2586356.039,293045739
2004,Catwoman,DC,3.2,10,21,16728411,6.21,2693785.99,293045739
2004,Spider-Man 2,Marvel,7.5,93,84,88156227,6.21,14195849.76,293045739
2004,The Punisher,Marvel,6.4,29,46.5,13834527,6.21,2227782.126,293045739
2005,Batman Begins,DC,8.3,85,84,48745440,6.41,7604592.824,295753151
2005,Elektra,Marvel,4.8,10,29,12804793,6.41,1997627.613,295753151
2005,Fantastic Four,Marvel,5.7,27,42,56061504,6.41,8745944.462,295753151
2006,Superman Returns,DC,6.3,76,69.5,52535096,6.55,8020625.344,298593212
2006,X-Men: The Last Stand,Marvel,6.8,57,62.5,102750665,6.55,15687124.43,298593212
2007,Fantastic Four: Rise of the Silver Surfer,Marvel,5.7,37,47,58051684,6.88,8437744.767,301579895
2007,Ghost Rider,Marvel,5.2,26,39,45388836,6.88,6597214.535,301579895
2007,Spider-Man 3,Marvel,6.3,63,63,151116516,6.88,21964609.88,301579895
2008,The Dark Knight,DC,8.9,94,91.5,158411483,7.18,22062880.64,304374846
2008,The Incredible Hulk,Marvel,7,67,68.5,55414050,7.18,7717834.262,304374846
2008,Iron Man,Marvel,7.9,94,86.5,98618668,7.18,13735190.53,304374846
2008,Punisher: War Zone,Marvel,6,26,43,4271451,7.18,594909.61,304374846
2009,Watchmen,DC,7.7,64,70.5,55214334,7.5,7361911.2,307006550
2009,X-Men Origins: Wolverine,Marvel,6.7,37,52,85058003,7.5,11341067.07,307006550
2010,Iron Man 2,Marvel,7.1,74,72.5,128122480,7.89,16238590.62,308745538
2010,Jonah Hex,DC,4.6,13,29.5,5379365,7.89,681795.3105,308745538
2011,Captain America: The First Avenger,Marvel,6.8,79,73.5,65058524,7.93,8204101.387,311591917
2011,Green Lantern,DC,5.9,27,43,53174303,7.93,6705460.656,311591917
2011,Thor,Marvel,7,77,73.5,65723338,7.93,8287936.696,311591917
2011,X-Men: First Class,Marvel,7.9,87,83,55101604,7.93,6948499.874,311591917
2012,Marvel's The Avengers,Marvel,8.7,92,89.5,207438708,7.92,26191756.06,314055984
2012,The Dark Knight Rises,DC,9.1,86,88.5,160887295,7.92,20314052.4,314055984
2012,Ghost Rider: Spirit of Vengeance,Marvel,4.5,17,31,22115334,7.92,2792340.152,314055984
2012,The Amazing Spider-Man,Marvel,7.6,74,75,62004688,7.92,7828874.747,314055984
```

INTRODUCING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 11: ITERATORS

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

11: ITERATORS

BY FRED ANNEXSTEIN, PHD

CHAPTER 11 OVERVIEW

In this module we will be covering the concepts associated with programming with iterators in Python. An iterator is an special object that contains a countable number of values. An iterator can be iterated upon, meaning that you can traverse through every value of the iterator with a simple for-loop. As we will see, in Python, an iterator is an object which implements the iterator protocol, which consist of the two simple methods called iter() and next().

CHAPTER 11 OBJECTIVES

By the end of this chapter, you will be able to...

1. Understand the concept and use cases of programming with iterators in Python.
2. Understand and apply the construction of iterators, and their use in several examples.
3. Understand and apply generator functions to the construction of iterators.
4. Understand the yield statement and the goals of lazy evaluation.
5. Apply the use of generators in support of lazy evaluation programming style.

ITERATORS IN PYTHON

Python provides a unified way to process the values or elements within a container sequentially; this special container is called an iterator. An iterator is an object that provides sequential access to values, one by one, and can be used by any for-loop.

By convention, an iterator has two components: a) a mechanism for retrieving the next element in the sequence being processed, and b) a mechanism for signaling that the end of the sequence has been reached and no further elements remain.

For any container, such as a list or range, an iterator can be obtained by calling the built-in iter() function. The contents of the iterator can be accessed by calling the built-in next() function. See the following example:

```
>>> primes = [23, 37, 51]

>>> type(primes)
list

>>> piter = iter(primes)

>>> type(piter)
list_iterator

>>> next(piter)
23

>>> next(piter)
37
```

```
>>> next(piter)
51

>>> next(piter)
Traceback (most recent call last):

  Input In [9] in <cell line: 1>
    next(piter)

StopIteration
```

We see from the above example that the next() function can be used to iterate through all the values in the list_iterator, and when there are no more elements in the container to iterate, the system throws a StopIteration exception.

An iterator can always be used with a 'for-in loop'. Python lists, tuples, dicts and sets are all examples of inbuilt iterators. These types are iterators because they implement two special methods. In fact, any container object that wants to be an iterator must implement the following two methods.

__iter__ is a method that is called for initialization of an iterator. This methods should return an object that has a next or __next__ method.

__next__ is a method that when called should return the next value for the collection.

When an iterator is used with a 'for-in' loop

loop construct, the for loop first initializes and then implicitly calls the next() function on the iterator object. This method should raise a StopIteration exception to signal the end of the iteration.

An iterator works by maintaining sufficient information for a local state to represent the current position in a sequence. Each time next is called, that position advances. Two separate iterators can track two different positions in the same sequence. However, two names for the same iterator will share a position, because as aliases they reference the same object.

```
>>> r = range(30, 100, 10)
>>> s = iter(r)  # 1st iterator over r
>>> next(s)
30
>>> next(s)
40
>>> t = iter(r)  # 2nd iterator over r
>>> next(t)
30
>>> next(t)
40
>>> u = t    #  u is alias for iterator t
>>> next(u)
50
>>> next(u)
60
```

Advancing the second iterator t does not

affect the first s. Since the last value returned from the first iterator was 40, it is positioned to return 50 next. On the other hand, the second iterator u=t is positioned to return 80 next.

```
>>> next(s)
50
>>> next(t)
70
```

Calling the iter() method on an iterator will return that iterator, not a copy. This behavior is included in Python so that a programmer can call iter() on a value to get an iterator without having to worry about whether it is an iterator or a simple container.

```
>>> v = iter(t)
# Another alias for the t iterator
>>> next(v)
80
>>> next(u)
90
```

The usefulness of iterators is derived from the fact that the underlying series of data for an iterator does not have to be represented and stored explicitly in memory. An iterator

provides a mechanism for considering each of a series of values in turn, but all of those elements do not need to be stored simultaneously. Instead, when the next element is requested from an iterator, that element may be computed on demand instead of being retrieved from an existing memory source.

RANGES ARE LAZY

Ranges are able to compute the elements of a sequence lazily because the sequence represented is uniform, and any element is easy to compute from the starting and ending bounds of the range. Thus in Python it is possible to compute functions such as len(), getitem(), and the in() methods extremely fast, as shown in the following:

```
r = range(1,10000000,7)

>>> 100004 in r
False
>>> 100003 in r
True
>>> len(r)
1428572
>>> r[987654]
6913579
```

Iterators allow for lazy generation of a much broader class of underlying sequential

datasets, because they do not need to provide access to arbitrary elements of the underlying series. Instead, iterators are only required to compute the next element of the series, in order, each time another element is requested. While not as flexible as accessing arbitrary elements of a sequence (called random access), sequential access to sequential data is often sufficient for data processing applications.

ITERABLES

Any object that can produce an iterator is called an *iterable* object. In Python, an iterable is anything that can be passed to the built-in iter() function. Iterables include sequence values such as strings and tuples, as well as other containers such as sets and dictionaries. Iterators are also iterables, because they have an iter() function attribute by definition.

Even unordered collections such as dictionaries must define an ordering over their contents when they produce iterators. Dictionaries and sets are unordered because the programmer has no control over the order of iteration. Python does guarantee certain properties about their order based on the hash function used in its specification.

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d
{'one': 1, 'three': 3, 'two': 2}
>>> k = iter(d)
>>> next(k)
'one'
>>> next(k)
'three'
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
3
>>> next(v)
2
>>> next(v)
Traceback (most recent call last):

  Input In [26] in <cell line: 1>
    next(v)

StopIteration
```

If a dictionary changes in structure because a key is added or removed, then all iterators become invalid and future iterators may exhibit arbitrary changes to the order their contents. On the other hand, changing the value of an existing key does not change the order of the contents or invalidate iterators.

```
>>> d.pop('two')
2
>>> next(k)
```

```
RuntimeError: dictionary changed size during
iteration
Traceback (most recent call last):
```

FOR-LOOP STATEMENTS

The for-in statement in Python operates on any iterator. Objects are iterable (an interface) if they have an __iter__ method that returns an iterator. Iterable objects can be the value of the <expression> in the header of a for statement:

```
for <name> in <expression>:
    <suite>
```

To execute a for-in statement, Python evaluates the header <expression>, which must yield an iterable object. Then, the __iter__ method is invoked on that value.

How does iter work? It calls the __iter__ magic method on its argument, so iter(something_iterable) is equivalent to something_iterable.__iter__(). So, any object is iterable if it implements this method, and this method must return an iterator.

How does next work? It calls the __next__ magic method on its argument, so next(the_iterator) is equivalent

to the_iterator.__next__(). So, to implement the iterator protocol, we must implement this method. This method eithers returns the next element in our iteration, or raises the StopIteration exception when there are no elements left.

```
>>> counts = [1, 2, 3]

>>> for item in counts:
        print(item)
1
2
3
```

In the above example, the counts list returns an iterator from its __iter__() method. The for statement then calls that iterator's __next__() method repeatedly, and assigns the returned value to item each time. This process continues until the iterator raises a StopIteration exception, at which point execution of the for statement concludes.

With our knowledge of iterators, we can implement the execution rule of a for statement in terms of basic while, assignment, and try statements.

```
>>> items = counts.__iter__()
>>> try:
        while True:
            item = items.__next__()
            print(item)
```

```
    except StopIteration:
        pass

1
2
3
```

Above, the iterator returned by invoking the __iter__ method of counts is bound to the variable name items so that it can be queried for each element in turn. The handling clause for the StopIteration exception does nothing, but handling the exception provides a control mechanism for exiting the while loop.

Python docs suggest that an iterator have an __iter__ method that returns the iterator itself, so that all iterators are iterable.

AN ITERABLE USER DEFINED TYPE

We illustrate using OOP classes to implement several iterators. Any class that will produce an iterable object must include the iter() and next() magic methods.

In this first example the Start10 class will construct an iterator that iterates starting from the number 10 and will iterate to any given input value, which is an argument to the object initializer.

```
class Start10:
     def __init__( self , limit):
         self.limit  = limit

     def __iter__( self ):
         self.x  = 9
         return self

     def __next__( self ):
         if self.x >=  self.limit:
             raise StopIteration
         self.x += 1 ;
         return self.x

>>> for i  in Start10(21):
       print (i, end = " ")
10 11 12 13 14 15 16 17 18 19 20 21

# will print nothing since 9 < 10
>>> for i  in Start10(9):
        print(i)
```

In this next example the Rev class will construct an iterator that iterates in reverse for any starting sequence given as an argument to the initializer. The reverse iterator works by keeping a pointer ptr that starts at the end of the sequence and moves backwards through the sequence.

```
class Rev:
  def __init__(self,seq):
    self.data = seq

  def __iter__(self):
    self.ptr = len(self.data)
```

```
    return self

  def __next__(self):
    if self.ptr == 0:
      raise StopIteration
    self.ptr -= 1
    return self.data[self.ptr]


>>> for i  in Rev(range(10)):
       print (i, end= " ")
9 8 7 6 5 4 3 2 1 0
```

In this next example the DoubleIt class will construct an iterator that takes another iterator as an argument, and then doubles (multiples by 2) any value iterated by the original iterator. The DoubleIt iterator works by using the iter() and next() methods of the original iterator.

```
class DoubleIt:
    def __init__(self, it):
      self.it = it
    def __iter__(self):
      iter(self.it)
      return self

    def __next__(self):
      return next(self.it) * 2
```

```
>>> for i in DoubleIt(iter([1, 2, 3, 4])):
        print(i, end= " ")

2 4 6 8

>>> for i in DoubleIt(iter(range(1,10,3))):
        print(i, end= " ")
2 8 14
```

GENERATORS AND YIELD STATEMENTS

With complex sequences, it can be quite difficult to program the __next__() method to save its place in the calculation. Python allows for an object called a generator that makes defining more complicated iterations easier by leveraging the features of the Python interpreter.

A generator is a special iterator that is returned by a special class of functions called a generator functions. Generator functions are distinguished from regular functions in that rather than containing return statements in their body, they use yield statements to return elements of a series.

Generators do not use attributes of an object to track their progress through a series. Instead, they control the execution of the generator function, which runs until the next yield statement is executed each time

the generator's __next__ method is invoked.

In the following example, we define a simple generator that mimics the operation of the Start10 iterable from above

```
def generator10(limit):
  print("Start10")
  x = 9
  while x < limit:
    x += 1
    yield x


>>> for i in generator10(21):
      print(i, end= " ")
Start10
10 11 12 13 14 15 16 17 18 19 20 21
```

In the following example we define a letters_generator, producing an iterator that contains all the letters. This example iterator shows that the implementation is much more compact using generators.

```
def letters_generator():
        current = 'a'
        while current <= 'z':
            yield current
            current = chr(ord(current)+1)

>>> for letter in letters_generator():
        print(letter, end=“ “)
a b c d e f g h i j k l m n o p q r s t u v
w x y z
```

Even though we never explicitly defined __iter__ or __next__ methods, the yield statement indicates to the interpretor that we are defining a generator function.

A generator function is different from a generator object. The generator function doesn't return a particular yielded value, but instead returns a generator object (which is a type of iterator).

The generator object has __iter__ and __next__ attribute methods, and each call to __next__ continues execution of the generator function from wherever it left off previously until another yield statement is executed.

The first time __next__ is called, the program executes statements from the body of the generator function until it encounters the yield statement. Then, it pauses and returns the value specifiec. The yield statements do not destroy the newly created environment, they preserve it for later. When __next__ is called again, execution resumes where it left off. The values of any bound names in the scope of generator function are preserved across subsequent calls to __next__.

We can walk through the generator by manually calling ____next__(). Here is an

example using the letters_generator function.

```
>>> letters = letters_generator()
>>> type(letters)
<class 'generator'>
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
‘d'
```

The generator does not start executing any of the body statements of its generator function until the first time __next__ is invoked. The generator raises a StopIteration exception whenever its generator function returns.

GENERATOR FOR NATURAL NUMBERS

In the next example we show how a generator can define a container with an infinite number of values. We use the example of the natural numbers 1,2,3,…

It is easy to define this infinite container by using a generator that only needs to store its current value i, and then with each call to next() increments the stored value.

```
def naturals():
  i = 1
  while True:
    yield i
    i += 1
```

INTRODUCING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 12: REGRESSION

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

12. REGRESSION

BY FRED ANNEXSTEIN, PHD

CHAPTER 12 OVERVIEW

In this module we will be covering an important method for data science called regression. We will consider methods of simple linear regression with both singular and multiple variable data sets. We will practice methods using several new libraries and new datasets, one based on time-series of city weather data, and another using state-level housing data.

CHAPTER 12 OBJECTIVES

By the end of this chapter, you will be able to...

Understand regression problems and solutions using time-series and geographic data.

Understand forecasting with simple and multiple linear regression.

Practice processing datasets for regression analysis.

Practice using the Scipy module for linear regression and visualizing the regression line using the Seaborn module.

Using regression for making prediction for city temperature changes over time.

Applying multiple linear regression to the problem of geographic data of housing

prices.

Practice using scoring methods for evaluating the modeling power of linear regression.

REGRESSION PROBLEMS

Regression is another important example of a supervised learning problem, one which has as input a data table with an additional attribute that we want to be able to predict. With solutions to regression problems the desired output consists of a mathematical model (or formula) using one or more continuous variables to predict the value of the target. A prototypical example of a regression problem would be the prediction of economic statistics, such as the next months inflation rate or the unemployment rate. In this chapter we will be working with temperature data for cities and geographic data on housing prices. We will be trying to find a linear model to predict future average monthly temperatures and the future of housing prices.

TIME SERIES

A time-seres is any data set of observations ordered in time. In one of our

running example problems in this chapter, the data that we'll use is a time series in which the observations or samples are monthly average temperatures for cities ordered by year.

The times series we process here are 'univariate time series' which have one observation per time unit. We look at univariate time-series for the average of the October high temperatures in Cincinnati, and the January high temperatures in New York City. We will process these time-series over several consecutive years. We wish to use these time-series to predict future temperatures. Since temperature varies by the season, our time series will only consist of one month per year, restricting our predictions to that single month.

More data can be incorporated in a prediction model using multivariate time series, which have two or more observations per sample time unit. Multivariate models could account for more weather information such as humidity and barometric pressure readings. To begin we will analyze a simple univariate time series. Later in the chapter we will address using a multivariate data set.

There are two major tasks often performed with time series, they are analysis and forecasting. Time series analysis looks at

existing time series data for patterns, with the goal of helping data analysts understand the aggregate data. A common analysis task is to look for seasonality in the data. This seasonality is surely to be found in the temperature time-series data of historical weather from cities.

FORECASTING WITH SIMPLE LINEAR REGRESSION

Using a technique called simple linear regression, we can make predictions on future (or historical) values. Simple linear regression finds the best-fitting line on the graph (or plot) of the dependent (or observation) and independent (or prediction target) variables. For our running example, the dependent variable is the month/year, and the target variable is the average high temperature for that calendar date. Simple linear regression describes the relationship between these variables with a straight line, and line that best-fitting for the data set is known as the regression line.

COMPONENTS OF THE SIMPLE LINEAR REGRESSION

The points along any straight line in two

dimensions, can be computed with the unique x,y-equation for that line:

```
 y = mx + b

where
• m is the line's slope,
• b is the line's intercept with the y-axis (at x = 0),
• x is the independent variable (the date in our example), and
• y is the dependent variable (the temperature in our example).
```

In simple linear regression, y = mx + b is the predicted value for any given x. Simple linear regression determines the two parameters, the slope (m) and intercept (b) of a straight line that best fits the data expressed as points in two dimensions (x,y). Hence, when we call a library function to perform a linear regression, then object returned should have two attributes - a slope and an intercept.

GETTING WEATHER DATA FROM NOAA

Let us now apply the regression method to the weather data for our regression study. The National Oceanic and Atmospheric Administration (NOAA) offers public historical data including time series for average high temperatures in specific cities over various

time intervals.

We obtained the data from NOAA's "Climate at a Glance" website:

https://www.ncdc.noaa.gov/cag/

We downloaded the October average high temperatures for Cincinnati from 1948 through 2022, and we also downloaded the January average high temperatures for New York city from 1895 through 2022.

LOADING THE AVERAGE HIGH TEMPERATURES INTO A DATAFRAME

Let's load and display the data from two .csv files using the method read_csv() from the Pandas library. Once read, we can look at the DataFrame's head and tail to get a sense of the structure of the data sets.

```
import pandas as pd

cincy= pd.read_csv('/Users/fred/Desktop/
cincy_1948_2022.csv')

nyc = pd.read_csv('/Users/fred/Desktop/
ave_hi_nyc_jan_1895-2022.csv')

>>> cincy.shape
(75, 3)

>>> nyc.shape
(124, 3)
```

```
>>> cincy.head()

     Date  Value  Anomaly
0  194810   52.1     -3.3
1  194910   59.3      3.9
2  195010   59.2      3.8
3  195110   57.7      2.3
4  195210   49.7     -5.7

>>> cincy.tail()

      Date  Value  Anomaly
70  201810   57.4      2.0
71  201910   59.6      4.2
72  202010   56.1      0.7
73  202110   61.7      6.3
74  202210   55.1     -0.3

>>> nyc.head()

   Date Value Anomaly
0 189501 34.2 -3.2
1 189601 34.7 -2.7
2 189701 35.5 -1.9
3 189801 39.6 2.2
4 189901 36.4 -1.0

>>> nyc.tail()

    Date Value Anomaly
119 201401 35.5 -1.9
120 201501 36.1 -1.3
121 201601 40.8 3.4
122 201701 42.8 5.4
123 201801 38.7 1.3
```

We wish to simplify the date column to make it just the year of the sample. Since the values are integers we can do an integer

divide by 100 to truncate the last two digits. Recall that each column in a DataFrame is a Series. Calling the Series method floordiv performs this integer division on every element of the Series, as seen as follows:

```
cincy.columns = ['Date', 'Temperature',
‘Anomaly']
cincy.Date = cincy.Date.floordiv(100)

nyc.columns = ['Date', 'Temperature',
‘Anomaly']
nyc.Date = nyc.Date.floordiv(100)
```

Let us look at the basic mean, standard deviation, and quartile statistics for the Temperature column.

```
>>> cincy.Temperature.describe()

count    75.000000
mean     55.769333
std       3.150460
min      48.500000
25%      53.850000
50%      55.900000
75%      57.550000
max      62.000000
Name: Temperature, dtype: float64

>>> nyc.Temperature.describe()
count    124.000000
mean      37.595161
std        4.539848
min       26.100000
25%       34.575000
```

```
50%       37.600000
75%       40.600000
max       47.600000
Name: Temperature, dtype: float64
```

LINEAR REGRESSION USING SCIPY

The SciPy (Scientific Python) library is widely used for applications in engineering, science and math. SciPy has a stats module which provides a function to do linear regression called linregress(). Here is how to import and execute this function on the cincy and nyc DataFrames. The object returned by linregress() has two attributes - a slope and an intercept as expected.

```
from scipy import stats

cincy_regress=
stats.linregress(x=cincy.Date,
                 y=cincy.Temperature)

>>> cincy_regress.slope
0.1814285714285704

>>> cincy_regress.intercept
52.33428571428573

nyc_regress= stats.linregress(x=nyc.Date,
                              y=nyc.Temperature)

>>> nyc_regress.slope
0.014771361132966163

>>> nyc_regress.intercept
```

```
8.694993233674289
```

The line defined by the slope and intercept pair gives us a simple way make predictions or forecasts of future temperatures. We can make a prediction for the year 2023, by plugging in to slope-intercept formula as follows:

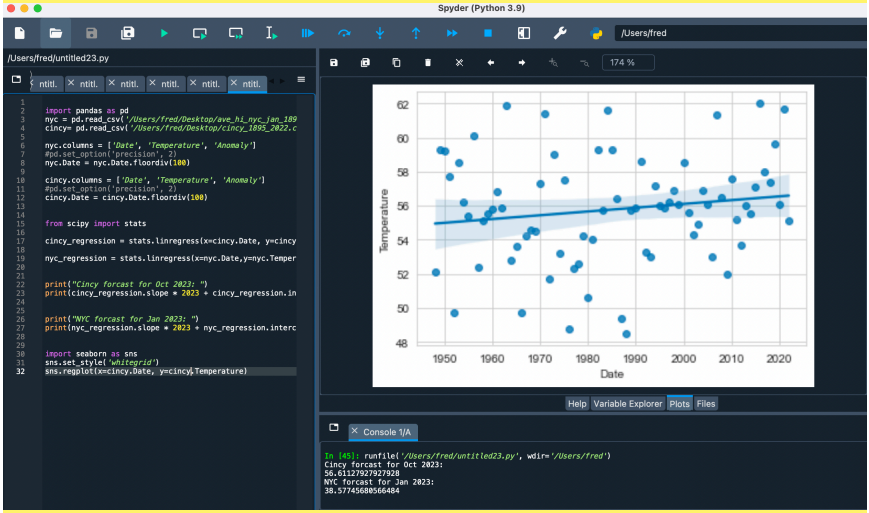
```
#Cincy forecast for avg high temp Oct 2023

>>> cincy_regress.slope * 2023 +
cincy_regress.intercept

56.6112792792792

# NYC forecast for avg high temp Jan 2023

>>> nyc_regress.slope * 2023 +
nyc_regression.intercept

38.57745680566484
```

DISPLAYING THE REGRESSION LINE USING SEABORN

We will use the Seaborn library to visualize the plot of the simple linear regression. To do this we use Seaborn's regplot function to plot each data point with the dates on the x-axis and the temperatures on the y-axis. The regplot function creates the scatter plot or scattergram in which the scattered blue dots

represent the Temperatures for the given Dates, and the straight line displayed through the points is the regression line.

```
import seaborn as sns
sns.set_style('whitegrid')
axes = sns.regplot(x=cincy.Date, y=cincy.Temperature)
```



The function regplot has x and y keyword arguments, which are expected to be one-dimensional arrays of the same length representing the x-y coordinate pairs to plot.

Recall that pandas automatically creates attributes for each column name if the name can be a valid Python identifier. Note the slope of the regression line in the plot below. It is clearly positive, showing that the trend is

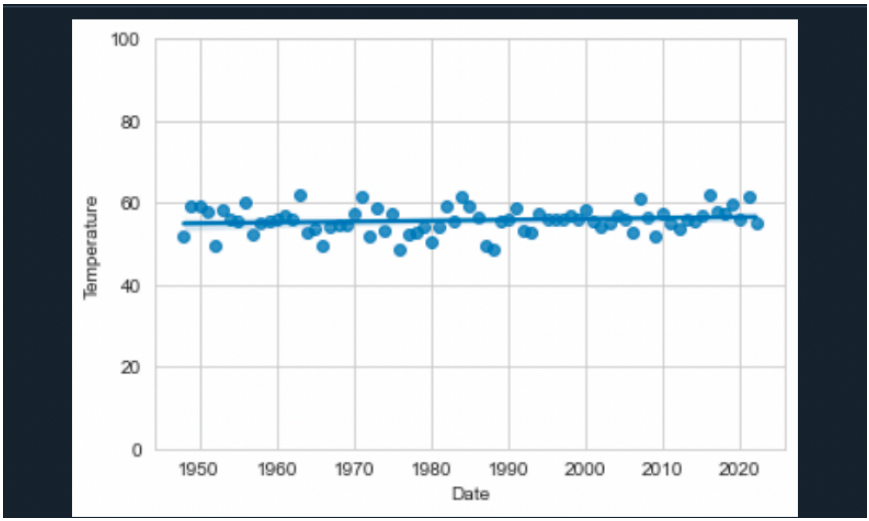
that temperature is rising by about 0.18 degrees per year for the cincy dataset.

SCALING FOR BETTER VISUALIZATION

The regression line's slope (lower at the left and higher at the right) indicates a warming trend over the last several years. In this graph, the y-axis represents a range of about 35-degrees. Thus the data appears to be spread significantly above and below the regression line, making it difficult to see the linear relationship between the x and the y variables. This is a common issue in data analytics visualizations, particularly when you have axes that reflect different kinds of data (i.e., dates and temperatures in this case).

In the preceding graph, this range is correlated to the height of the graph. The Seaborn and Matplotlib modules auto-scale the axes, based on the data's range of values. However, we can scale the y-axis range of values to emphasize the linear relationship. Here, we scaled the y-axis from a 35-degree range to a 100-degree range (from 0 to 100 degrees). An we show the resulting plot which has a more linear character.

```
axes = sns.regplot(x=cincy.Date,
y=cincy.Temperature)
axes.set_ylim(0, 100)
```



PREDICTING WHEN CERTAIN TEMPERATURE WILL BE REACHED

Assuming that the linear trend continues, we can ask the question what year do we expect the average October temperature in Cincinnati reach 70 degrees Fahrenheit. Of course we can use algebra to solve.

```
70 = m Year + b, so

Year = 70 – b / m
```

```
#Script for the Linear Regression Solution

import pandas as pd
cincy= pd.read_csv('/Users/fred/Desktop/
cincy_1948_2022.csv')
cincy.columns = ['Date', 'Temperature', '
Anomoly']
cincy.Date = cincy.Date.floordiv(100)

slope = cincy_regress.slope
intercept = cincy_regress.intercept

print("The Year for 70 for Oct in Cincy:" ,
(70 – intercept) / slope )

The Year for 70 for Oct in Cincy:
2627.2803458311932
```

An alternative is to run a while loop along the regression line starting from this year 2022.

```
thisyear = 2022

temp = slope * thisyear + intercept
while temp < 70.0:
    thisyear += 1
    temp = slope * thisyear + intercept

print(thisyear) # prints value 2628
```

REGRESSION WITH SKLEARN

The sklearn library gives us more powerful

tools to practice with linear regression, and can potentially be used to improve the model through more rigorous testing.

To carry out linear regression we will use the LinearRegression estimator from sklearn.linear_model module. By default, this estimator uses all the numeric features present in a dataset. Here, we perform simple linear regression using one feature as the independent variable. So, we'll need to select the Date feature from the dataset. When you select one column from a two-dimensional DataFrame, the result is a one dimensional Series.

However, scikit-learn estimators require their training and testing data to be two-dimensional arrays, or two-dimensional array-like data, such as lists of lists or Pandas DataFrames. To use one-dimensional data with an estimator, you must transform it from one dimension containing n elements, into two dimensions containing n rows and one column as you'll see below.

As we did in the previous case study, let's split the data into training and testing sets. Note the use of the keyword argument random_state, this is used for the purposes of reproducibility.

TRAINING THE MODEL

Sklearn does not have a separate class for simple linear regression. For sklearn we train a LinearRegression estimator as follows.

LinearRegression estimator computes the best fitting regression line for the data. The LinearRegression estimator iteratively adjusts to optimize the choice of slope and intercept values so as to minimize the sum of the squares of the distances of the data from the line.

The optimal slope is stored in the estimator's coeff_ attribute (m in the equation) and the optimal intercept is stored in the estimator's intercept_ attribute (b in the equation).

Note that the values obtained are somewhat different from the Scipy lingress function above, since we are using for the training dataset a subset of the original data.

```
from sklearn.linear_model import
LinearRegression

regress = LinearRegression()
regress.fit(X=X_train, y=y_train)

>>> regression.coef_
array([0.02116697])
```

```
>>> regress.intercept_
13.69890898941808
```

COMPARING SCIPY TO SKLEARN

Here is a script which shows the differences between using SciPy and Sklearn versions of Linear Regression. Sklearn is designed to create training and testing using random subsets of the data. Therefore the models will differ somewhat based on the random subset selected for training, which we show now.

```
import pandas as pd
cincy= pd.read_csv('/Users/fred/Desktop/
cincy_1948_2022.csv')
cincy.columns = ['Date', 'Temperature', '
Anomoly']
cincy.Date = cincy.Date.floordiv(100)


from scipy import stats
cincy_regress=
stats.linregress(x=cincy.Date,
                 y=cincy.Temperature)
slope = cincy_regress.slope
intercept = cincy_regress.intercept
print("SciPy Regression Solution:", slope,
intercept)



from sklearn.model_selection import
train_test_split
import random
```

```
X_train, X_test, y_train, y_test =
train_test_split(
    cincy.Date.values.reshape(-1, 1),
cincy.Temperature.values,
random_state=random.randint(1,10))


from sklearn.linear_model import
LinearRegression
skregress = LinearRegression()
skregress.fit(X=X_train, y=y_train)
print("Sklearn Regression Solution:",
skregress.coef_, skregress.intercept_ )

Output:

SciPy Regression Solution:
0.02215647226173543 11.788735893788505

Sklearn Regression Solution:
[0.02116697] 13.69890898941808
```

RESHAPING SERIES FOR TRAINING

Note that in the code above there several uses of the expression cincy.Date, which returns the Series object for the Date column. The Series values attribute returns a NumPy array containing that Series values.

To train we must transform this one dimensional array into two dimensions. To do this we call the array's reshape method. Normally, we pass two arguments which are the precise number of rows and columns. However, when the first argument is -1, this tells the reshape to infer the number of rows,

based on the number of columns and the number of elements in the original array. The transformed array will have only one column.

We can confirm the 75%–25% train-test split by checking the shapes of X_train and X_test, as follows:

```
>>> X_train.shape
(56, 1)
>>> X_test.shape
(19, 1)
```

TESTING THE MODEL

Let us now test the model using the data in X_test and check some of the predictions throughout the dataset by displaying the predicted and expected values for every fifth element, which we do by slicing the two arrays and zipping them together into a single object z:

```
predicted = regress.predict(X_test)
expected = y_test
z = zip(predicted[::5], expected[::5])

for p, e in z:
    print(f'predicted: {p:.2f}, expected:
{e:.2f}')

Output:
predicted: 55.26, expected: 54.20
predicted: 55.49, expected: 53.20
```

```
predicted: 56.09, expected: 53.30
predicted: 55.93, expected: 49.40
```

MULTIPLE LINEAR REGRESSION

With multiple linear regression we assume that we have more than one observed dependent variable. The model still assumes that the relationship between the dependent variable y and multiple dependent variables is linear. That is the target y is modeled as a linear sum of the dependent variables, each with their own computed coefficient.

To illustrate multiple linear regression we will process the California Housing dataset bundled with sklearn which has 20,640 samples, each with eight numerical features. We'll perform a multiple linear regression that uses all eight numerical features to make more sophisticated predictions of housing prices. We expect that the predictions will be better than if we were to use only a single feature or a subset of the features.

We will test this hypothesis in the lab exercises, where we will ask you to perform simple linear regressions with each individual features, and compare the results with the multiple linear regression computed here.

This housing dataset was derived from the 1990 U.S. census, using one row per census

group. The dataset has 20,640 samples—one per group—with eight features each:

```
Eight Features in the Housing Dataset:

• median income—in tens of thousands
• median house age – has a max of 52
• average number of rooms
• average number of bedrooms
• block population
• average house occupancy
• house block latitude
• house block longitude
```

Each sample also has as its target a corresponding median house value in hundreds of thousands of dollars, so a value of 3.55 would represent $355,000. In the dataset, the maximum value for this feature is 5, which represents a maximum house value of $500,000 - which is probably a serious problem of data accuracy.

Let's load the dataset and familiarize ourselves with it's structure. The fetch_california_housing function from the sklearn.datasets module returns a Bunch object containing the data and other metadata information about the dataset. For example, the string indices of the columns is stored in feature_names attribute, as seen as follows:

```
from sklearn.datasets import
fetch_california_housing
```

```
cali = fetch_california_housing()

>>> cali.feature_names
['MedInc',
'HouseAge',
'AveRooms',
'AveBedrms',
'Population',
'AveOccup',
'Latitude',
‘Longitude']
```

Next, let’s create a DataFrame from the Bunch’s data, target and feature_names arrays. The first snippet below creates the initial DataFrame using the data in cali.data and with the column names specified by cali.feature_names. The second statement adds a column for the median house values stored in cali.target:

```
cali_df = pd.DataFrame(cali.data,
     columns=cali.feature_names)

cali_df['MedHouseValue'] =
pd.Series(cali.target)
```

Once again we will use sklearn’s train_test_split to prepare the data for the purposes of training and testing the model.

```
from sklearn.model_selection import
train_test_split
```

```
X_train, X_test, y_train, y_test =
train_test_split( cali.data, cali.target,
random_state=11)

>>> X_train.shape
(15480, 8)
>>> X_test.shape
(5160, 8)
```

Now we call the estimator and attempt to fit the data with a linear regression:

```
from sklearn.linear_model import
LinearRegression

regress = LinearRegression()

regress.fit(X=X_train, y=y_train)
```

AN ITERABLE FOR THE REGRESSION SLOPE COEFFICIENTS

Multiple linear regression produces separate coefficients for the slope of each feature (stored in coeff_) and one intercept (stored in intercept_). Here we display these by using the built-in enumerate method used to create an iterable container for all the coefficient values:

```
>>> regress.intercept_
-36.88295065605547

e = enumerate(cali.feature_names)
for i, name in e:
   print(f'{name:>10}: {regress.coef_[i]}')
```

```
MedInc: 0.4377030215382206
HouseAge: 0.009216834565797713
AveRooms: -0.10732526637360985
AveBedrms: 0.611713307391811
Population: -5.756822009298454e-06
AveOccup: -0.0033845664657163703
Latitude: -0.419481860964907
Longitude: -0.4337713349874016
```

For positive coefficients, we can say that the median house value increases as the feature value increases. For example, the positive coefficient for number of bedrooms show a positive correlation with median home values. For negative coefficients, the median house value decreases as the feature value increases, for example as happens with the average number of bedrooms in a block group.

Note that the population coefficient has a negative exponent (e-06), so the coefficient's value is actually very close to zero, so a block group's population apparently has little effect the median house value. WIth multiple linear regression we use these coefficient values to make predictions with the following linear equation:

```
y = m1 x1 + m2 x2 + … mn xn + b
where
• m1, m2, …, mn are feature coefficients,
• b is the intercept,
```

```
• x1, x2, …, xn are the feature values (that
is, the values of the independent
variables),
and
• y is the predicted value (that is, the
dependent variable)
```

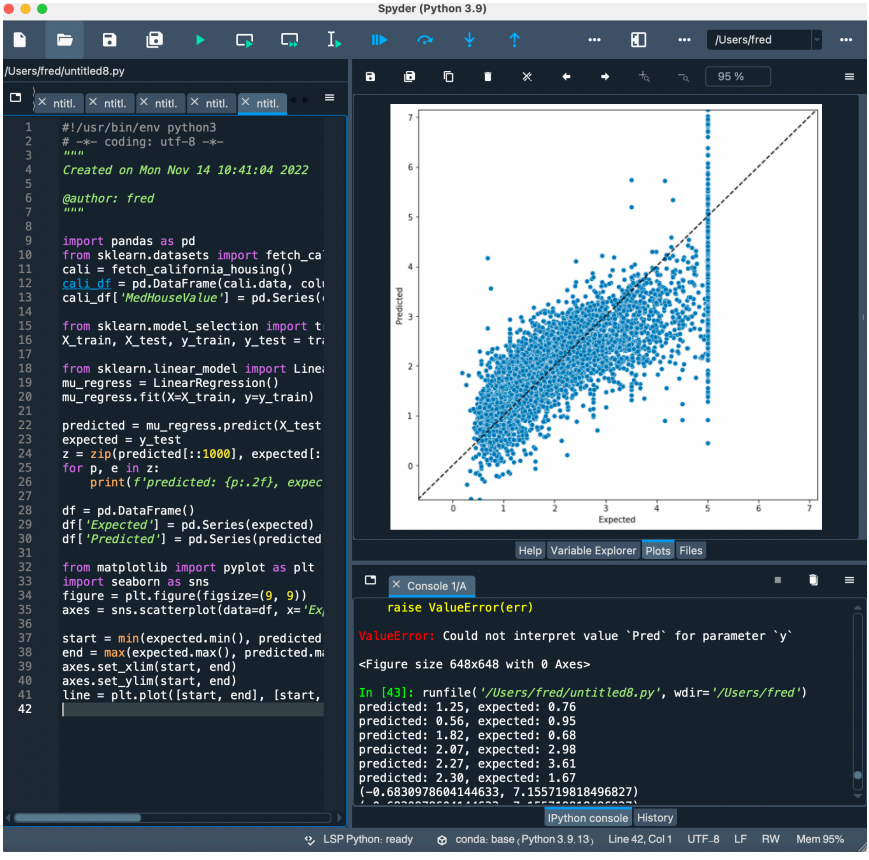
We use the same logic as above to list predictions by slicing the datasets.

```
predicted = regress.predict(X_test)
expected = y_test
z = zip(predicted[::1000], expected[::1000])

for p, e in z:
    print(f'predicted: {p:.2f}, expected:
{e:.2f}')

Output:
predicted: 1.25, expected: 0.76
predicted: 0.56, expected: 0.95
predicted: 1.82, expected: 0.68
predicted: 2.07, expected: 2.98
predicted: 2.27, expected: 3.61
predicted: 2.30, expected: 1.67
```

VISUALIZING THE EXPECTED VS. PREDICTED PRICES

Let us look at the expected vs. predicted median house values for the test data. First, let's create a DataFrame containing columns for the expected and predicted values. Next we plot the data as a scatter plot with the expected (target) prices along the x-axis and the predicted prices along the y-axis. Next we set the x- and y-axes' limits to use the same scale along both axes. Finally we plot a

line x=y that represents perfect predictions (note that this is not a regression line). The following displays a line between the points representing the lower-left corner of the graph (start, start) and the upper-right corner of the graph (end, end). The third argument ('k--') indicates the line's style. The letter k represents the color black,and the -- indicates that plot should draw a dashed line:



```
# Script to Visualize the the Expected vs.
# Predicted Prices using Multiple Linear
# Regression Housing Price Estimator

import pandas as pd
from sklearn.datasets import
fetch_california_housing
cali = fetch_california_housing()
cali_df = pd.DataFrame(cali.data,
columns=cali.feature_names)
cali_df['MedHouseValue'] =
pd.Series(cali.target)

from sklearn.model_selection import
train_test_split
X_train, X_test, y_train, y_test =
train_test_split( cali.data, cali.target,
random_state=11)

from sklearn.linear_model import
LinearRegression
mu_regress = LinearRegression()
mu_regress.fit(X=X_train, y=y_train)

predicted = mu_regress.predict(X_test)
expected = y_test
z = zip(predicted[::1000], expected[::1000])
for p, e in z:
    print(f'predicted: {p:.2f}, expected:
{e:.2f}')

df = pd.DataFrame()
df['Expected'] = pd.Series(expected)
df['Predicted'] = pd.Series(predicted)

from matplotlib import pyplot as plt
import seaborn as sns
figure = plt.figure(figsize=(9, 9))
axes = sns.scatterplot(data=df,
x='Expected', y='Predicted')

start = min(expected.min(), predicted.min())
```

```
end = max(expected.max(), predicted.max())
axes.set_xlim(start, end)
axes.set_ylim(start, end)
line = plt.plot([start, end], [start, end],
'k--')
```

REGRESSION MODEL METRICS

Sklearn provides many metrics functions for evaluating how well estimators predict results and for comparing estimators to choose the best one(s) for your particular study. These metrics vary by estimator type. Among the many metrics for regression estimators is the model's R2 score. To calculate an estimator's R2 score, call the sklearn.metrics module's r2_score function with the arrays representing the expected and predicted results:

R2 scores range from 0.0 to 1.0 with 1.0 being the best possible. Negative R2 scores are possible and indicate that the chosen model fits the data really poorly. An R2 score of 1.0 indicates that the estimator perfectly predicts the dependent variable's value, given the independent variable(s) value(s). An R2 score of 0.0 indicates the model cannot make predictions with any accuracy, based on the independent variables' values.

Another common metric for regression models is the mean squared error, which

calculates the difference between each expected and predicted value—this is called the standard mean squared error, which squares each difference and then calculates the average of the squared values. The mean_square_error is alway positive or zero, in the case there is no errors. Here are examples of the two metrics described:

```
>>> from sklearn import metrics

>>> metrics.r2_score(expected, predicted)
0.6008983115964333

>>> metrics.mean_squared_error(expected,
predicted)
0.5350149774449119
```

Both metrics give us a way of ranking or comparing estimators and their quality. It is best when the mean squared error value is closest to 0 and the R2 score closest to 1.0.

LAB FOR CHAPTER 12

For this lab we will practice with simple linear regression with the California Housing Dataset. In the previous sections we showed how to perform multiple linear regression for predicting housing prices using the California Housing dataset for both training and testing.

If a dataset has meaningful features for

predicting the target values, and you have the choice between running simple and multiple linear regression, then you will generally choose multiple linear regression, since using more data will generally result in better predictions. As we have seen when using sklearn's LinearRegression estimator, the method uses all the numerical features of the data by default thus performing multiple linear regression.

In this lab exercise you'll perform single linear regressions with each feature individually and compare the prediction results to the multiple linear regression we have carried out previously.

To do so, first split the dataset into training and testing sets, then select one feature, as we did with the DataFrame in this chapter's simple linear regression case study. Train the model using that one feature and make predictions as you we have done in the multiple linear regression case study. Do this for each of the eight features. Print out each of the simple linear regression's R2 score and mean squared error score.

Produce a table that can be used to determine which feature produced the best results. Write a short explanation of how the simple linear regression estimators compare with the multiple linear regression. Upload

your code and table for this assignment.

Here is an example output table which prints out for each feature in the data the R2 score and the MSE score.

```
Multiple Linear Regression using All features
R2 score : 0.6008983115964333
MSE score: 0.5350149774449118


Feature 0 has R2 score : 0.4630810035698606
          has MSE score 0.7197656965919478
Feature 1 has R2 score : 0.013185632224592903
          has MSE score 1.3228720450408296
Feature 2 has R2 score : 0.024105074271276283
          has MSE score 1.3082340086454287
Feature 3 has R2 score : -0.0011266270315772875
          has MSE score 1.3420583158224824
Feature 4 has R2 score : 8.471986797708997e-05
          has MSE score 1.3404344471369465
Feature 5 has R2 score : -0.00018326453581640756
          has MSE score 1.340793693098357
Feature 6 has R2 score : 0.020368890210145207
          has MSE score 1.3132425427841639
Feature 7 has R2 score : 0.0014837207852690382
          has MSE score 1.3385590192298276All
```

INTRODUCING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 13: MONTE CARLO SIMULATION

BY FRED ANNEXSTEIN, PHD

PYTHON PROGRAMMING

13. MONTE CARLO SIMULATION

BY FRED ANNEXSTEIN, PHD

CHAPTER 13 OVERVIEW

In this module we will be covering an important algorithmic method for data science called the Monte Carlo method. Monte Carlo methods rely on access to a fast and statistically sound random number generators. Python provides such a generator in the random module. We will consider a variety of problems that we can apply Monte Carlo methods to achieve accurate simulation and estimation results.

CHAPTER 13 OBJECTIVES

By the end of this chapter, you will be able to... .

Understand the use cases of the Monte Carlo methods

Understand the reliance of such methods on effective random number generators.

Understand how to use seed values for reproducibility.

Understand how to use Monte Carlo methods for various ball and bin simulations.

PSEUDO-RANDOM NUMBERS

Several Python libraries will produce

pseudorandom number generators (PRNG). The standard modules random and numpy have implementations of efficient pseudo-random number generators for various distributions. For integers, there is a generator for uniform selection from a range. For sequences, there is generator for uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement, as others.

Almost all module functions depend on the basic function random(), which generates a random float uniformly in the range [0.0, 1.0). Python uses the Mersenne Twister algorithm as the core generator. This produces 53-bit precision floats and has an enormous period of 2**19937-1. The Mersenne Twister is one of the most extensively tested random number generators in existence.

A COMPUTATIONAL EXPERIMENT

Let us run an experiment with random number generator random(). We will compute the mean and the variance, which is the average of the squared deviations from the theoretical mean of 0.5. Our experiment uses N=10,000 random samples and

calculates the sample mean and sample variance.

```
import random
sum, sumdev = 0.0,0.0
N= 10000
for i in range(N):
    next = random()
    sum += next
    dev = (next - 0.5)**2
    sumdev += dev
mean = sum / N
print(f"Mean:{mean}")
var = sumdev/(N-1)
print(f”Variance:{var}")

Output:
Mean:0.49933762112884134
Variance:0.08385481699150024
```

Theory tells us that for a standard uniform distribution U(0,1) of a random variable between 0 and 1, the expected value is 1/2 and the variance is 1/12. These are close to the computed sample values.

SEEDING THE RANDOM-NUMBER GENERATOR FOR REPRODUCIBILITY

The function random() is deterministic and generates pseudorandom numbers, based on an internal calculation that begins with a numeric value known as a seed. Repeatedly calling random produces a sequence of

numbers that appear to be random, because each time you start a new interactive session or execute a script that uses the random() function, Python internally uses a different seed value. When you are debugging logic errors in programs that use randomly generated data, it can be helpful to use the same sequence of random numbers until you have eliminated the logic errors, before testing the program with other values.

To do this, you can use the random module's seed function to seed the random-number generator yourself—this forces random() to begin calculating its pseudorandom number sequence from the seed you specify.

In the next example, we use the int() function along with random() to generate a random integer number between 1 and 6 to simulate a dice roll. The output will be the same for each run, since we fixed the seed value.

```
from random import random,seed
seed(43)
for roll in range(20):
    print(1 + int(random()*6) , end=' ‘)

Output:
1 5 1 3 5 5 3 3 1 3 3 6 4 5 3 2 6 5 3 1
```

MONTE CARLO ESTIMATION OF PI

One simple demonstration of the Monte Carlo algorithm is the estimation of the number Pi. Here we use the idea of sampling random (x, y) points in a 2-D unit square using a standard uniform distribution. All the points in this square whose distance from the origin is at most 1.0 lies inside or on top the quarter-circle. For a Monte Carlo estimation of pi we can calculate the ratio of the number points that fall inside the circle versus the total number of generated points. This ratio will tend to pi/4 since we are using a quarter of a circle with radius 1, that is quarter of a circle whose area is Pi. Here is the code to carry out this Monte Carlo estimation:

```
from random import random

N = 10000
circle_hits = 0
for i in range(N):
  x,y = random(), random()
  if x**2 + y**2 <=1:
    circle_hits +=1


pi_est = 4 * circle_hits / N
print("Final Estimate of Pi=", pi_est)

Output:
Final Estimation of Pi= 3.1424
```

MONTE CARLO INTEGRATION

Integration or summing the area under a curve lends itself to solution by Monte Carlo techniques. This is possible since we can view integration as calculating the average value of a function along an interval [a,b] and then multiplying by the length of the interval which is simply b-a.

Let us consider integrating the sin() function. In the following code we choose a large number of random x values in the interval [a,b] and compute the sum of the function values at those points. This large sum yields an average value for the function, and thus provides the integral.

The example below estimates the integral of the sin() function for the range between a=0 and b=np.pi =3.141592653589793. Since math says that integrating sin is -cos, and so the Monte Carlo estimate shows correctness to 3 decimal places.

```
import numpy as np
from random import random

# limits of integration are a, b
a = 0
b = np.pi # gets the value of pi
N = 10000

integral = 0.0
```

```
def f(x):
    return np.sin(x)

for i in range (N):
    x  = a + (b-a)*rng()
    integral += f(x)

ans = (b-a) * integral/N
print (f"{ans} is the estimated value.”)

Output:
2.0040349814696534 is the estimated value.
```

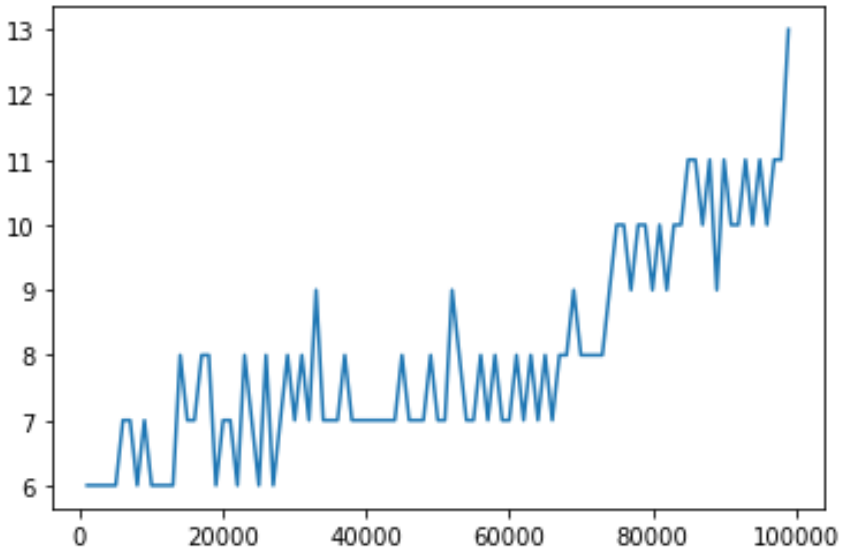
BALLS AND BIN ESTIMATION

There are many examples in Computer Science application we model collision problem. It is popular to use what is called balls and bins analysis to investigate such collision problems. Let us consider a container with N bins, and we throw balls at random, each ball falling into any one bin with equal chance. We would like to know how many balls will be in the largest bin after throwing N balls at random. Surely the number is greater than 1 and less than N, but where in that range is the answer likely to be.

We run a Monte Carlo simulation to answer that question. In the following code we run experiments for each N up to 1,000,000. To speed up the simulation we only pick values of N divisible by 100. We will produce a line

plot of the results. To do this we store a list maxbin values for the size of the largest bin in each experimental value of N.

```
import numpy as np
from random import random

balls = np.arange(100,100000,100)
maxbin = []
for N in balls:
  bins = np.zeros(N)
  for b in range(N):
    bins[int(N * random())] +=1
  maxbin.append(max(bins))

import matplotlib.pyplot as plt
plt.plot(balls, maxbin)
plt.show()
```

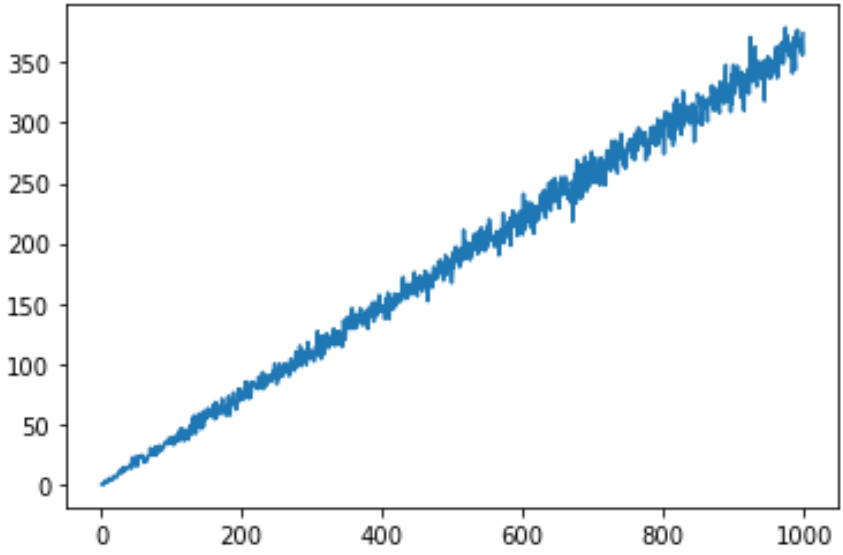


We see from the output that throwing N balls into N bins never causes any bin to be

loaded with more than 13 balls, and the larges bins are never smaller than 6 balls.

MODULE 13 LAB

For this lab you will run a Monte Carlo simulation for a different balls and bins problem. For this problem we are interested in knowing the number of empty bins when N balls are thrown at random into N bins. For this assignment you are to write code to run a simulation for each N from 1 to 1000, and plot the results using the matplotlib plot function. When you do this you should see a striking linear relationship in the output.



We would like to know the slope of this

this linear relationship. Write code to run a linear regression to find the best fit line for your dataset. Your output should be similar to the following:

```
SciPy Linear Regression Solution
 slope: 0.36837836634229415
 intercept: -0.02602000798390236
 rvalue: 0.997749733854947
```

Notice that the rvalue is very close to 1, and the slope is very close to the number 1/e = 0.36787944117. We leave this as an optional exercise to show why that is so.