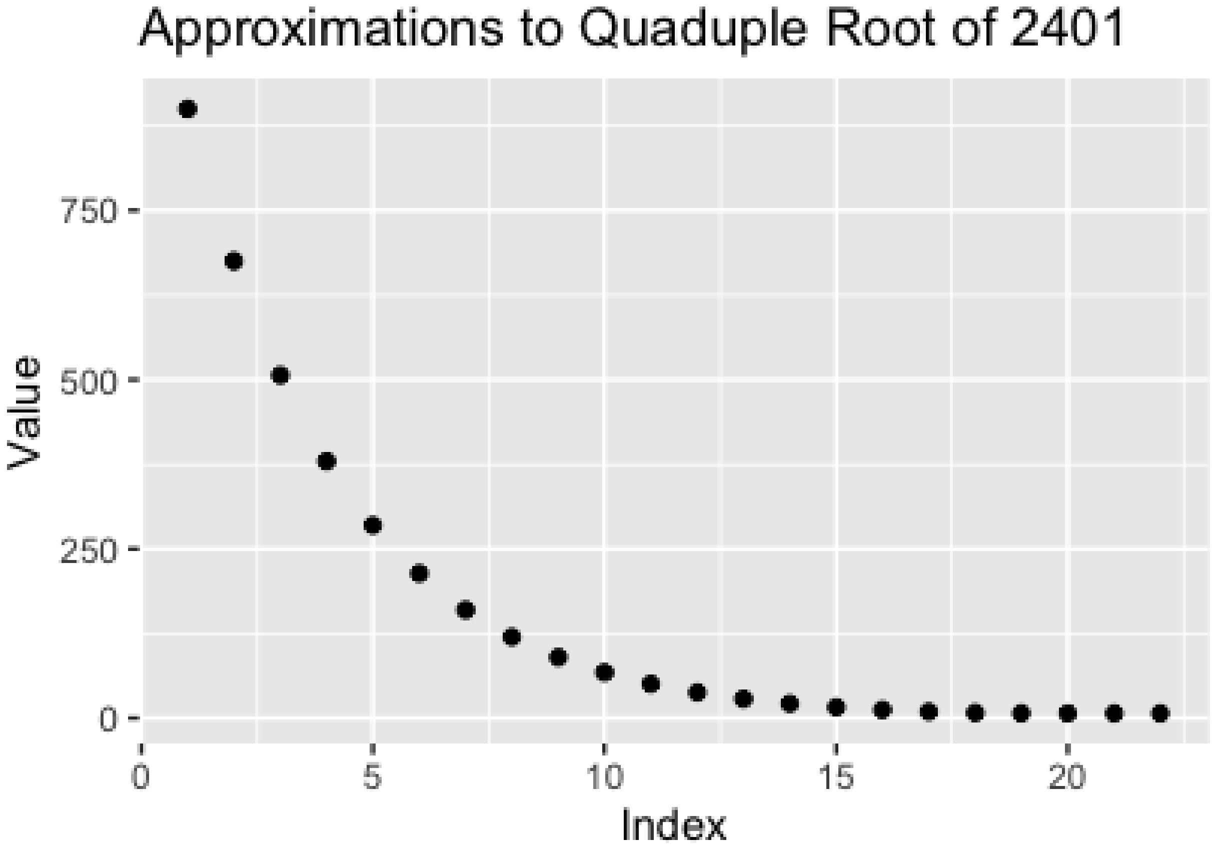
Quadruple Roots - Mathematical loops to find cube roots

How to compute : $\sqrt[4]{x}$?

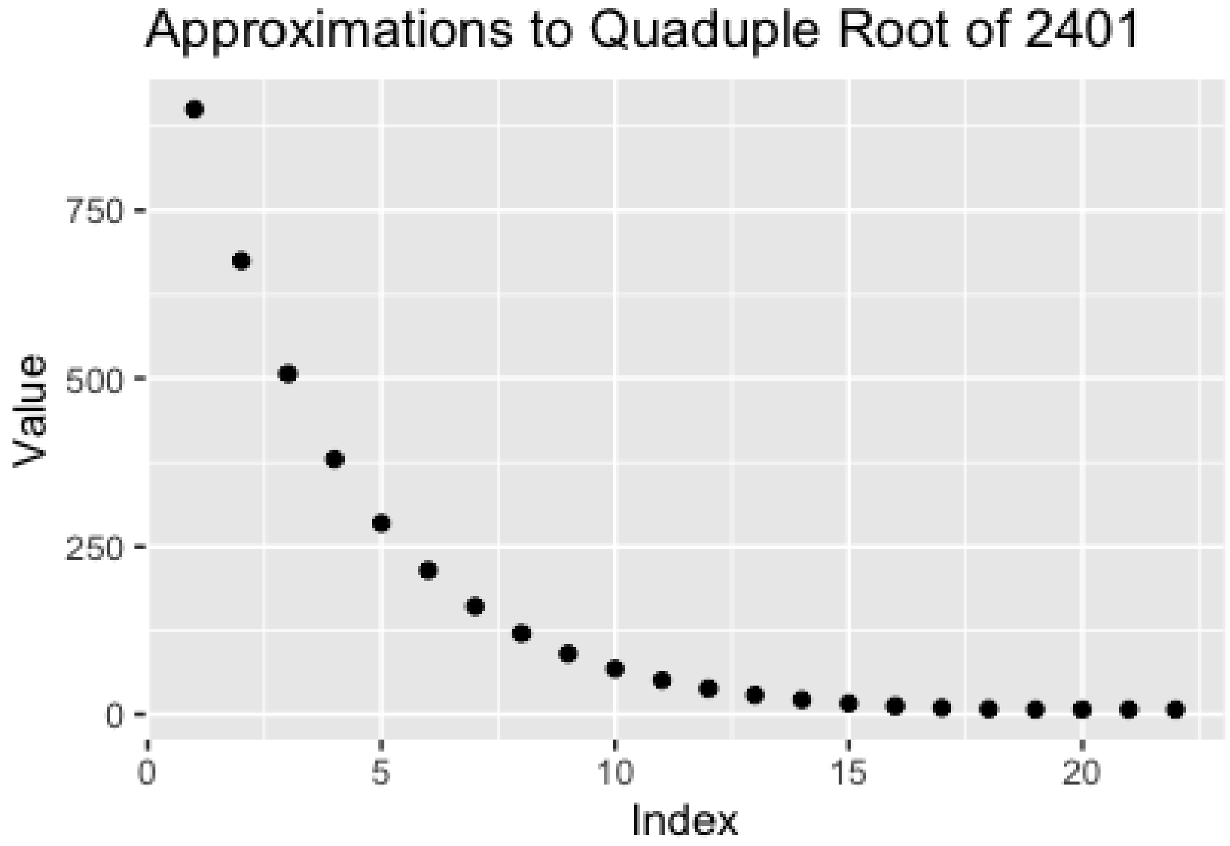
Method

The function initializes the guess for the quadruple root and iteratively refines it using an approximation formula until the approximation is within the specified tolerance.

Quadruple Roots - Approximations to find quadruple roots; result = 7



Quadruple Roots - Approximations to find quadruple roots; result = 7



900.375000346933,
675.2812510825596,
506.46094026121705,
379.8457098164694,
284.8842933147822,
213.66324594739956,
160.24749599845413,
120.18576786629791,
90.13967165836334,
67.60557331037666,
50.70612258852655,
38.03419610727984,
28.5365566959587,
21.428247703754217,
16.132191727838684,
12.242116142093641,
9.508748977118197,
7.82973307939813,
7.122821698405513,
7.00314043464742,
7.000002111777238,
7.000000000000956

Quadruple Roots - Code

```
def quadruple_root_approximation(number, tolerance=1e-6):
    # Initial guess for the fourth root
    guess = number / 2.0
    # Iterate until the approximation
    # is within the specified tolerance
    while abs(guess**4 - number) > tolerance:
        # Update the guess using the approximation formula
        guess = (3 * guess + number / (guess**3)) / 4.0
        print(f" guess = {guess}")
    return guess

# Example: Calculate the fourth root of 2401
input_number = 2401
result = quadruple_root_approximation(input_number)

# Display the result
print(f"The fourth root of {input_number}")
print(f" is approximately: {result}")
```

Quadratic Roots – The Problem Defined

To Solve: $x^2 + 3x - 4 = 0 \quad (1)$

Want to have roots

$$x_1 = ?\, and\ x_2 = ?$$

Quadratic Root Calculation

Quadratic Equation:

$$\mathrm{a}x^2 + bx + c = 0 \quad (2)$$

Quadratic Formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (3)$$

Special Note

Note the $x_{1,2}$ to imply that there are two solutions (i.e., x_1 and x_2) to find for a second-degree equation as observed from the x^2.

Programmed Solution

```
def calc_quad_eqn_roots(
a: float, b: float, c: float) -> float:
        """Calculate roots of quadratic equation."""
        D = (b * b - 4 * a * c) ** 0.5
        x_one = (-b + D) / (2 * a)
        x_two = (-b - D) / (2 * a)
        return x_one, x_two

print(f"{calc_quad_eqn_roots(1,2,1)}")
```

- Three floating-point inputs: a, b, and c
- Two floating-point outputs: x_{one} and x_{two}
- How does it calculate the roots of a quadratic equation?

Discrete Structures!

CMPSC 102

Programming Constructs

ALLEGHENY COLLEGE

Key Questions & Learning Objectives

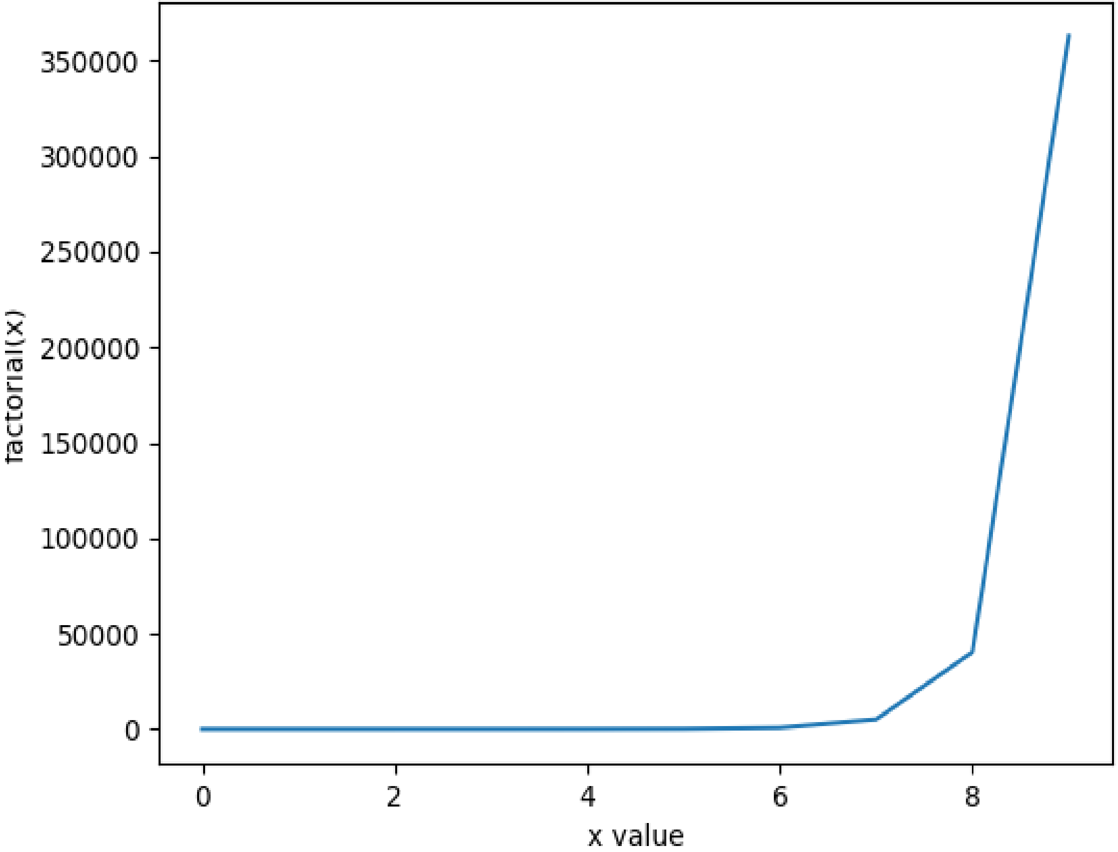
- How do I use non-recursive functions, recursive functions, and lambda expressions to perform mathematical operations such as computing the absolute value of a number and the means of a sequence of numbers?

- To remember and understand some discrete mathematics and Python programming concepts, setting the stage for exploring of discrete structures.

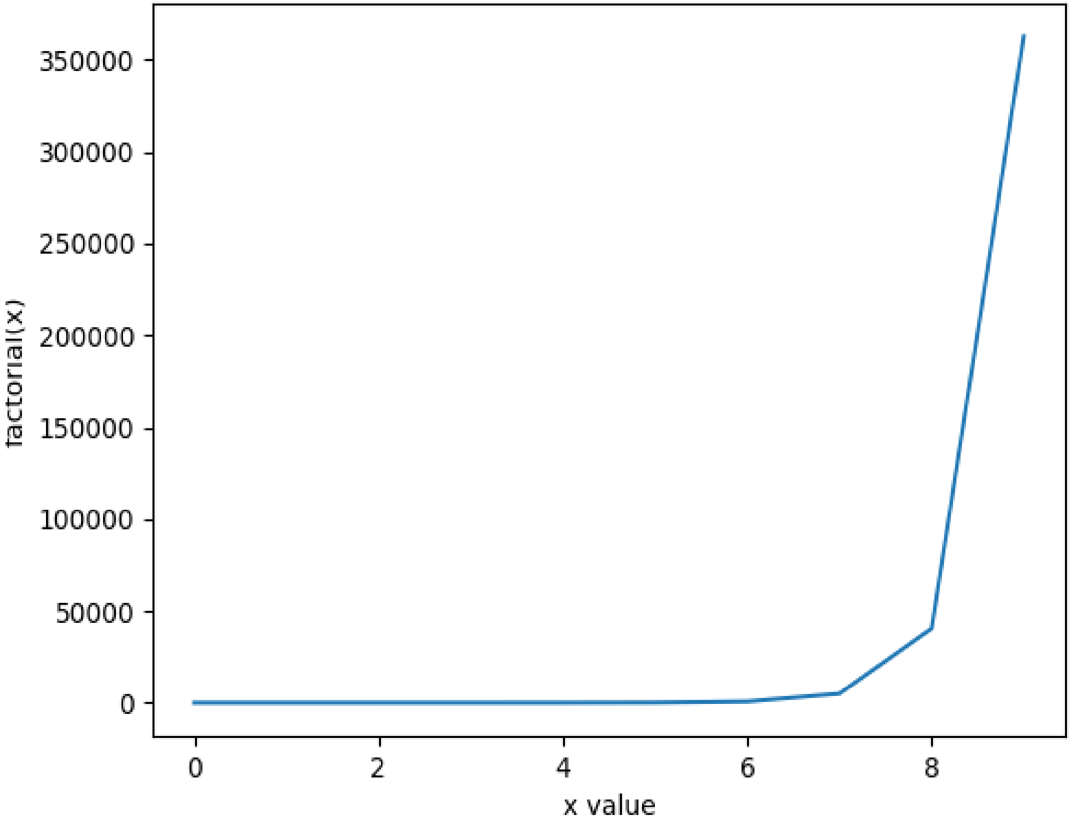
Python Programming Retrospective

- Python code is designed to be **intuitive**
- Key components of Python programming include:
    - Function and their definitions
    - Input parameters for functions
    - The code block that completes the function's work
    - Return statements
    - Invocations of functions (calls to functions)
    - Collecting the returned values (function outputs).
- Investigate the ways to make the above commands possible with definitions and call using Python.

Factorials - values get quickly get big



Factorials - values get quickly get big



x	$fac(x)$
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800

Plotting factorials – use Jupyter for this code!

```
import matplotlib.pyplot as plt
import math

# get factorial data
x_list = [i for i in range(10)]
factorials_list = [math.factorial(x) for x in x_list]
print("x,factorial(x)")
# formatting data
for i in range(len(x_list)):
        xvalue_int = x_list[i]
        fvalue_int = factorials_list[i]

# prepare plot
print(f"x values :{x_list}")
print(f"factorial(x) : {factorials_list}")
plt.plot(x_list, factorials_list)
plt.xlabel('x value')
plt.ylabel('factorial(x)')
plt.show()
```

Factorials

Factorials: one definition

$$N! = \pi_{i=1}^{N} i = 1 * 2 *..* (N-1) * N$$

Factorials: another definition

$$N! = \frac{(N+1)!}{(N+1)} = \frac{(N+1) * N!}{(N+1)}$$

Factorials are applied to integers

Factorials

Factorials

$$N! = N * (N-1) * (N-2) * \cdots * (2) * (1)$$

$$5! = 5 * 4 * 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

$$3! = 3 * 2 * 1$$

$$2! = 2 * 1$$

$$1! = 1$$

$0! = 1$ (Special case by convention)

Factorials defined

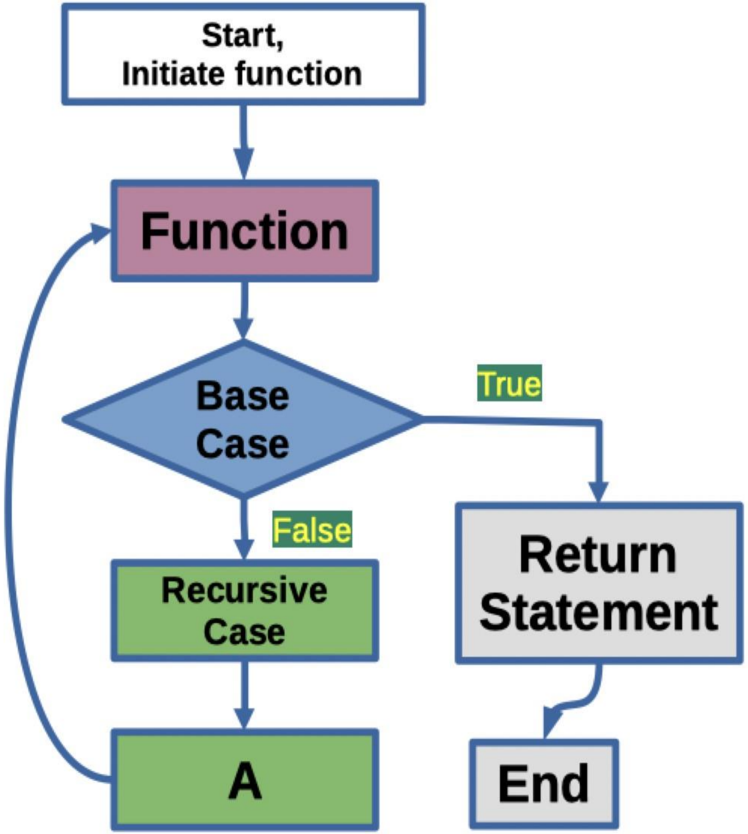
$$N! = [(N-1)! + (N-2)!] * (N-1)$$

$$7! = (6! + 5!) * 6$$

$$6! = (5! + 4!) * 5$$

$$5! = (4! + 3!) * 4$$

Creating Solutions



Calculating Factorials by Recursion

```
def factorial(number: int):
        if number == 1:
                return 1
        return number * factorial(number - 1)

num = 5
print("The factorial of " + str(num)
        + " is " + str(factorial(num)))
```

- The recursive *factorial* function calls itself!
- How does this function ever stop executing?
- What are the benefits to using recursive functions?

Calculating Factorials by Recursion

```
def factorial(number: int):
        if number == 1:
                return 1
        return number * factorial(number - 1)

num = 5
print("The factorial of " + str(num) +
        " is " + str(factorial(num)))
```

- Where is the base case?
- Where is the recursive case?
- How could this code work without these two functions?

Recursive Factorial Function - To consider

- As an equation: $n! = n \times (n-1) \times (n-2) \times \ldots \times 1$
    - What are the **parts** of a recursive function in Python?
    - Defined by **cases** using conditional logic (*a case to go, and one to force a stop*)
    - A mathematical function defined to **call itself**
    - A recursive call that makes progress to a **base case**
    - A **base case** that **stops** the **recursive function calls**
    - Repeatedly perform an operation through (*self*) function calls
    - What would happen if you input a **negative number**?
- How could you write this function with **iteration**?

A Solution Using Basic Conditions - No numbers less than zero

```
def factorial(number: int):
        if number == 1:
                return 1
        if number < 0: #Catch negative numbers
                print("cannot compute")
        if number > 1:
                return number * factorial(number - 1)

num = -5
print("The factorial of " + str(num)
+ " is " + str(factorial(num)))
```

A Solution Using While - No numbers less than zero

```
def factorial(number: int):
    while number > 0:
        if number == 1:
            return 1
        if number > 1:
            return number * factorial(number - 1)
    print("cannot compute")
```

What Can YOU Do With Higher-Order Functions

You can pass a **function** as an **argument** to a **function**!

Why Do We Care About Higher-Order Functions!?

- Supports general-purpose function creation
- Allows executable functions as function input
- Supports both code reuse and modularity

Higher-Order Functions - library declaration and square()

Functions that allow another function as a parameter

```
from typing import Callable

# define a function that can square a number;
# use print statements for the purposes of debugging so that the behavior of this
# function and the next function are made clear

def square(number: int) -> int:
        print(f"Called square({number})")
        print(f" returning {number*number}")
        return number * number

print(square(5))
```

Higher-Order Functions - Call_twice() with execution code

```
# define a higher-order function that can accept a function
# as input and a number as input and then call the provided
# function with the provided input; again, use print
# statements for the purposes of debugging so that the
# behavior of this function is made clear

def call_twice(f: Callable[[int], int], number: int) -> int:
        print(f"Calling twice {f} with number {number}")
        return f(f(number))

# execution
num = 5

# give function and function's parameter
result = call_twice(square, num)

print("Calling the square twice with "
+ str(num) + " is " + str(result))
```

Higher-Order Functions

```
def square(number: int):
        print(f"Called square({number})")
        print(f" returning {number*number}")
        return number * number
```

- The behavior of **higher-order** functions in Python:
- square() is a function computes number*number and returns value.

Higher-Order Functions

```
def call_twice(f, number: int):
        print(f"Calling twice {f} with number {number}")
        return f(f(number))
```

- call_twice() is a function that calls a function f twice
- First, call_twice() calls f with number
- Then, call_twice() calls f with f(number)
- Finally, call_twice() returns result of f(f(number))
- Can you predict the output of the call_twice() function?
- How would you test the call_twice() function? Can you express it differently?

Higher-Order Functions

Calling twice <function square at 0x104c30940> with number 5
Called square(5)
 returning 25
Called square(25)
 returning 625
Calling the square twice with 5 is 625

Lambda Expressions - Also known as, “anonymous functions”

```
def call_twice(f, number: int):
        print(f"Calling twice {f} with number {number}")
        return f(f(number))
```

```
square = lambda x: x*x
number = 5
result = call_twice(square, number)
print("Calling square lambda twice " +
        "with " + str(number) +
        " is " + str(result))
```

- Functions are values in the Python programming language
- square is an expression that has a function as its value

Lambda Expressions

```python
def call_twice(f, number: int):
    print(f"Calling twice {f} with number {number}")
    return f(f(number))
```

```python
square = lambda x: x*x
number = 5
result = call_twice(square, number)
print("Calling square lambda twice " +
      "with "  +  str(number) +
      " is " + str(result))
```

```
Calling twice <function <lambda> at 0x37500c8> with number 5
Calling square lambda twice with 5 is 625
```

- Lambda functions are known as anonymous functions and add simplicity in programming
- Useful for small function input to other functions

Discrete Structures!

CMPSC 102

Setting Up Projects

ALLEGHENY COLLEGE

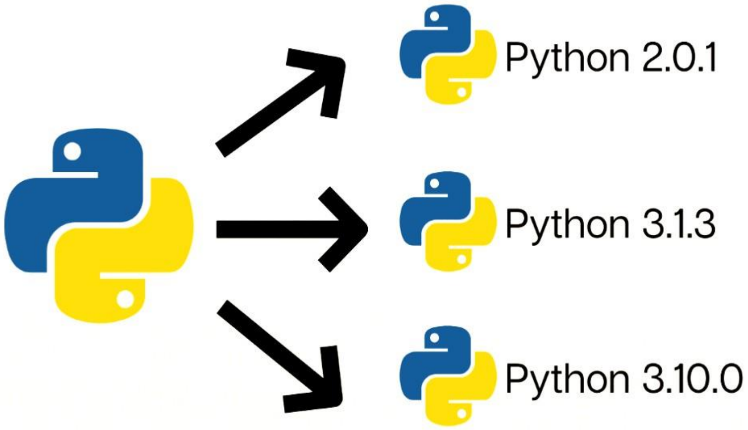
Key Questions and Learning Objectives

- How do I use virtual environments like Venv and Poetry, along with tools like Typer and other resources, to create a professional Python project?

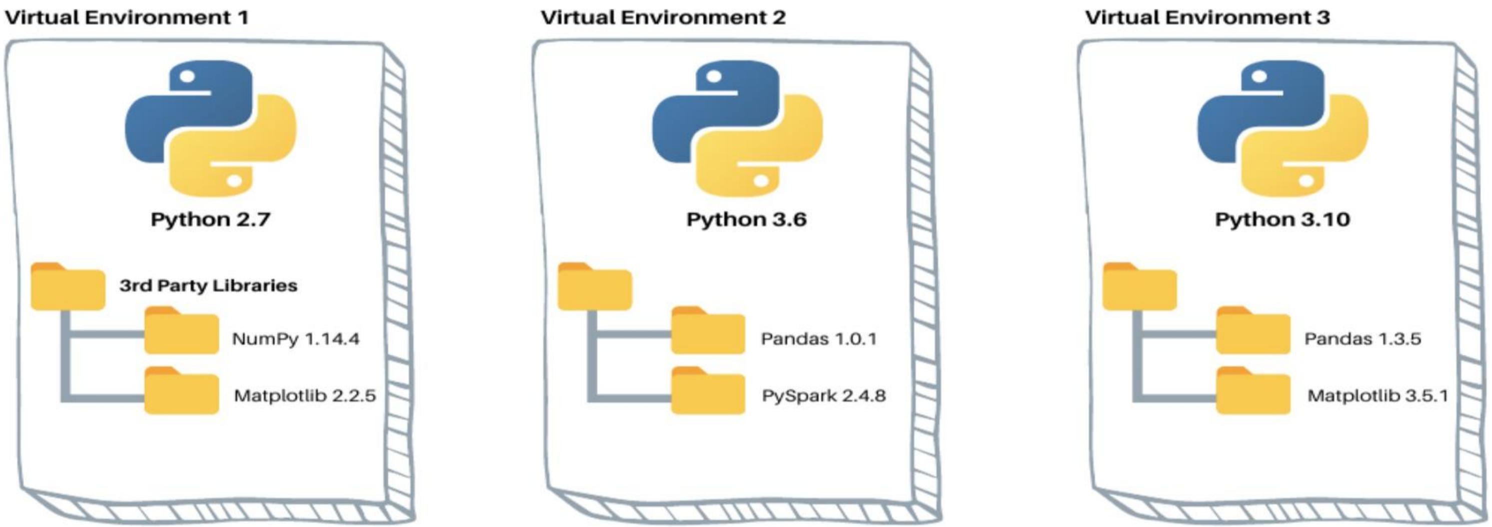
- To learn how to use libraries and dependencies for development with Python code and programming techniques to create the foundations for a professional project.

Virtual Environments

- Projects' dependencies and specific versions of libraries.
- Not all projects require the same dependencies; How do you mix projects on your computer?



Go Virtual



- Virtual environments maintain specific libraries and dependencies projects
- Shipping software: build an exact copy of development environment on client's machine to use software.

Regression Analysis Project

- Decide on the purpose and composition of the project
  - Our project: a regression analysis demonstration from SciKit-Learn
    - https://scikit-learn.org/stable/supervised_learning.html#
  - No command line parameters
  - No output, other than screen printing
  - Execution: Program complete regression analysis of random values
  - One function in project: main()
  - **Dependencies**: scikit-learn, numpy, seaborn

Setting Up Virtual Environment

- Create a project directory

```
mkdir projects
cd projects
```

- Create virtual environment using Python

```
python3 -m venv myenv
# see the file tree
find . -not -path '*/\.*'
```

- Activate myenv the virtual environment

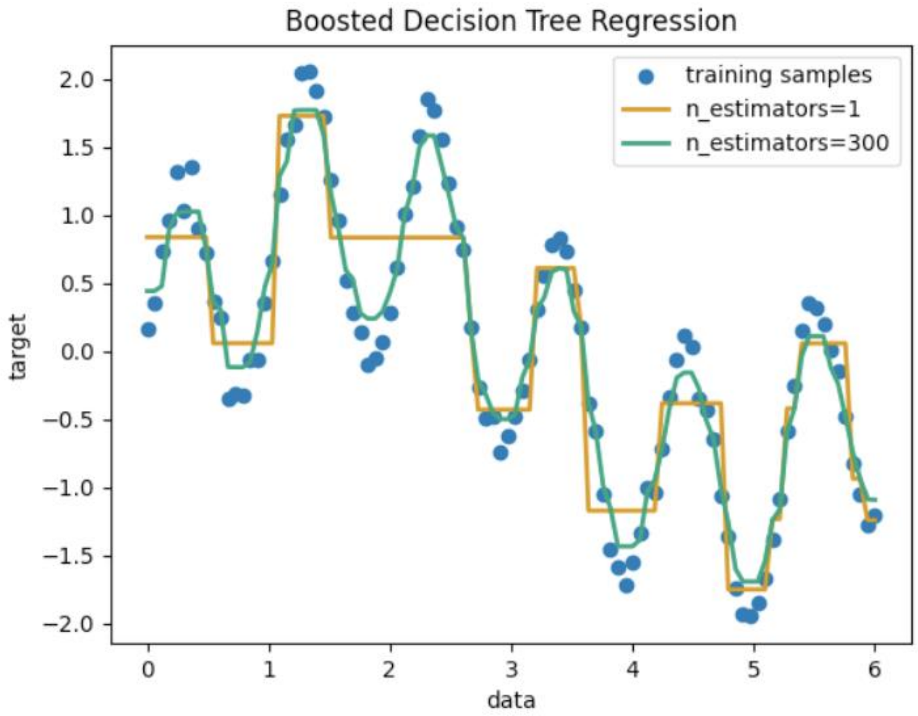
```
source myenv/bin/activate  # macOS/Linux
myenv\Scripts\activate     # Windows
```

- Install Dependencies

```
pip install numpy
pip install seaborn
pip install scikit-learn
# or try: pip install sklearn
```

Output From Executing the Script

- Execute given Script in myenv : `Python3 sciKitDemo.py`



Black: a Python Script formatter

Got Clutter?

- How to maintain *readable* code?
- How to reduce white-space in code to improve readability?

- A code formatting resource
- https://black.readthedocs.io/en/stable/getting_started.html

Black: a Python Script formatter

- Install Dependencies

```
pip install black
```

- Linting example: As a String to Printed Line

```
black --code "print ( 'hello, world'       )"
```

- Linting example: Standard Input to File

```
echo "print ( 'hello, world' )" | black - > out.txt
```

https://black.readthedocs.io/en/stable/usage_and_configuration/the_basics.html

- We will use this with Poetry (up next)

We Need Poetry!!

Work without Hope, by Samuel Taylor Coleridge Lines Composed 21st February 1825

All Nature seems at work. Slugs leave their lair The bees
are stirringbirds are on the wing
And Winter slumbering in the open air, Wears on his
smiling face a dream of Spring! And I the while, the
sole unbusy thing,
Nor honey make, nor pair, nor build, nor sing.

Yet well I ken the banks where amaranths blow,
Have traced the fount whence streams of nectar flow.
Bloom, O ye amaranths! bloom for whom ye may, For
me ye bloom not! Glide, rich streams, away!
With lips unbrightened, wreathless brow, I stroll: And
would you learn the spells that drowse my soul? Work
without Hope draws nectar in a sieve,
And Hope without an object cannot live.

A Bigger Virtual Environment

*Poetry is **a tool for dependency management and packaging** in Python. It allows you to declare the libraries your project depends on, and it will manage (install/update) them for you. Poetry offers a **lockfile** to ensure repeatable installs and can build your project for distribution.*

Python Resource - Poetry

PYTHON PACKAGING AND DEPENDENCY MANAGEMENT MADE EASY

Poetry

https://python-poetry.org/

- Management support for Python and its resources
- Environments: manage dependencies in isolation
- Package: create a stand-alone executable application
- Publish: expedite and simplify the release of program to PyPI

Python Resource - Typer

https://typer.tiangolo.com/

- Command line interface support for program inputs and parameters
- Annotations: assigns types to functions that accept arguments (parameters)
- Productivity: types aid in the creation of the interface
- Checking: Confirm that inputs match expected types.

A Hello User Program

- Decide on the purpose and composition of the project
  - Our project: Say hello to the user
  - Parameters firstName, middleName and lastName as parameters
  - No output, other than screen printing
  - Execution: Program greets user by full name
  - One function in project: main()

Setup Steps

- Make a working directory

```
mkdir project2
cd project2
```

- Use Poetry to create new project

```
poetry new hello_user
cd hello_user
```

- Add Project Dependencies

```
poetry add typer
poetry add rich
```

- Add Project Development Dependencies

```
poetry add -D black mypy
```

Mypy: http://mypy-lang.org/

Setup Steps

- Add File: project2/hello user/hello user/ init .py

```
"""Package-level docstring for hello_user package."""

__version__ = "0.1.0"
```

- Add File:projects/hello user/pyproject.toml

```
[project] ...

[tool.poetry.scripts]
hello_user = "hello_user.main:cli"

[tool.poetry.group.dev.dependencies] ...
```

- Update Poetry

```
poetry install
```

Add File: projects/hello user/hello user/main.py - File located in sanbox: main.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from rich.console import Console
import typer

# create a Typer object to support the command-line interface
cli = typer.Typer()
@cli.command()
def main(first: str = "", middle: str = "", last: str = ""):
        """Say hello to the person having a name of first, middle and last name"""
        console = Console()
        console.print(" Hello to;")
        console.print(f"\t First = {first}")
        console.print(f"\t Middle = {middle}")
        console.print(f"\t Last = {last}")
# end of main()
```

Basic Reformatting with Black

```
poetry run black hello_user tests
```

```
/Users/hangzhao/.local/pipx/venvs/poetry/lib/python3.9/site-packag
LibreSSL 2.8.3'. See: https://github.com/urllib3/urllib3/issues/30
  warnings.warn(
reformatted hello_user/moreFun.py
reformatted hello_user/main.py

All done! ✨ 🍰 ✨
2 files reformatted, 2 files left unchanged.
hangzhao@Mac hello_user %
```

Execute Project

What do you see?

```
# run from projects/hello_user/hello_user
poetry run python3 hello_user/main.py --help
```

```
Usage: main.py [OPTIONS]

Say hello to the person having a name of first, middle and last name

╭─ Options ─────────────────────────────────────────────────────────────────────────────────────
│ --first                      TEXT
│ --middle                     TEXT
│ --last                       TEXT
│ --install-completion         [bash|zsh|fish|powershell|pwsh]  Install completion for the specified shell. [default: None]
│ --show-completion            [bash|zsh|fish|powershell|pwsh]  Show completion for the specified shell, to copy it or customize the installation. [default: None]
│ --help                                                        Show this message and exit.
╰───────────────────────────────────────────────────────────────────────────────────────────────
```

Execute Project

- What do you see?

```
# run from projects/hello_user
poetry run hello_user
```

- Without parameters

```
poetry run hello_user
Hello to;
            first =
            middle =
            last =
```

Execute Project

- What do you see?

```
# run from projects/hello_user poetry run hello_user

--first John

--middle H.

--last Davis
```

- Without parameters

```
Hello to;
        first = John
        middle = H.
        last = Vader
```