CSCI 127

James Goudy

December 2, 2025

Contents

0.1 Welcome to CSCI 127 Joy and Beauty of Data

These are the class notes for CSCI 127 Joy and Beauty of data. This course focuses on core programming techniques applicable to all languages along with specific libraries and techniques only to Python.

0.2 Introduction

Origins:

- **Successor to ABC:** In the late 1980s, Guido van Rossum, while working at CWI in the Netherlands, envisioned Python as an improvement on the ABC programming language. ABC was known for its clarity, but lacked features like exception handling, which Python incorporated.
- **Focus on Readability:** A core principle for Python was code readability. Python's syntax emphasizes clear, concise statements, making it easier for programmers to understand and maintain code. This focus helped make Python popular for teaching programming concepts.

Naming:

- **Monty Python Inspiration:** The name Python has no connection to the snake. Instead, it's a homage to the British comedy group Monty Python's Flying Circus, a favorite of van Rossum.

Uses Today:

- **Versatility is Key:** Python's strength lies in its versatility. It's considered a general-purpose language, meaning it can be applied to various tasks. Here are some prominent areas:
  - **Web Development:** Python frameworks like Django and Flask power many websites and web applications.
  - **Data Science and Machine Learning:** Python excels in data analysis and manipulation. Libraries like NumPy, Pandas, and scikit-learn make it a go-to choice for data scientists and machine learning engineers.
  - **Scripting and Automation:** Python's ability to automate tasks efficiently makes it valuable for automating repetitive processes across different domains.
  - **Scientific Computing:** Python offers powerful libraries like SciPy and Matplotlib for scientific calculations, visualizations, and simulations.
- **Popularity and Growth:** Python consistently ranks among the most popular programming languages. Its ease of use, vast ecosystem of libraries, and active developer community continue to fuel its growth.

0.2.1 Comprehensive overview of Python

(encompassing its strengths, weaknesses, and uses:)

Description:

Python is a high-level, general-purpose programming language known for its readability and ease of use. Created in the late 1980s by Guido van Rossum, it emphasizes clear syntax and utilizes indentation to define code blocks, making it resemble natural language. This focus on readability makes Python a popular choice for beginners and for collaborative development projects.

Strengths:

- **Readability:** Python's clean syntax and focus on indentation make code easy to understand and maintain. This is a major advantage for large projects or when working with multiple programmers.
- **Versatility:** Python's strength lies in its ability to be applied to various tasks. It's a general-purpose language that can be effectively used for web development, data science, scripting, and scientific computing.
- **Interpreted Language:** Unlike compiled languages that require conversion to machine code before running, Python code is interpreted line by line at runtime. This allows for faster development cycles as changes can be tested immediately.
- **Large Standard Library:** Python includes a rich set of pre-written functions and modules for common tasks, saving developers time and effort.

- **Extensive Third-Party Libraries:** Beyond the standard library, Python boasts a vast ecosystem of third-party libraries that cater to specialized needs in various domains. Popular examples include NumPy and Pandas for data science, Django and Flask for web development, and scikit-learn for machine learning.

Weaknesses:

- **Performance:** Compared to compiled languages like C++ or Java, Python can be slower due to its interpreted nature. This might not be a major concern for many applications, but it's a factor for performance-critical systems.
- **Memory Management:** Python relies on automatic garbage collection, which can sometimes lead to inefficiencies in memory usage. While generally good, for memory-intensive tasks, other languages might offer more control.
- **Dynamic Typing:** Python is dynamically typed, meaning variable types are determined at runtime. While convenient, it can also lead to runtime errors if data types aren't what you expect. Statically typed languages can help catch these errors earlier in the development process.
- **Not Ideal for Mobile Development:** While Python can be used for mobile app development with frameworks like Kivy, it's generally not the preferred language due to performance considerations and limitations in native mobile development libraries compared to languages designed for mobile development.
- **Security:** While Python itself isn't inherently insecure, its dynamic nature can make it more vulnerable to certain types of security vulnerabilities compared to statically typed languages. Careful coding practices and security considerations are essential when using Python for security-sensitive applications.

Applications:

- **Web Development:** Python frameworks like Django and Flask streamline web development by providing structure, tools, and libraries for building web applications.
- **Data Science and Machine Learning:** Python's strength in data manipulation and analysis, coupled with powerful libraries like Pandas, NumPy, and scikit-learn, makes it a preferred choice for data scientists and machine learning engineers.
- **Scripting and Automation:** Python excels in automating repetitive tasks, saving time and reducing human error. It can be used to automate data processing, file management, web scraping, and more.
- **Scientific Computing:** Scientific libraries like SciPy and Matplotlib provide advanced functionalities for numerical computations, data analysis, and creating scientific visualizations.

Overall:

Python's strengths often outweigh its weaknesses, making it a great choice for many programming tasks. It's a versatile and beginner-friendly language with a vast ecosystem of libraries and a supportive community. However, it's important to be aware of its limitations, such as performance considerations for speed-critical applications, to make informed decisions when choosing the right programming tool for the job.

0.2.2 Where is Python used?

- Artificial Intelligence and Machine Learning
- Data Analytics
- Game Programming
- Applications
- Web Development
- Finance
- Sciences

0.3 Readings

These are reading links to online textbooks and other material.

0.3.1 The Basics

Chapter 1 - thinkcspy
Chapter 2 - thinkcspy

0.3.2 Functions

Chapter 5 - Modules - thinkcspy
Chapter 6 - Functions - thinkcspy

End Of Topic

0.4 The Basics

Definition
Python - is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level builtin data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python.org

0.4.1 Editors

- Idle
- Spyder / Anaconda
- PyCharm
- VSCode - must also load appropriate plugins
- others....

Run in interactive mode

```
# first program
print ("hello world")
```

0.4.2 Coding Concepts

- Case sensitive
- Space-based - tabbing has meaning
  - List appropriate use of white space and tabbing
- Do not go more than 80 characters
- Escape characters \n (new line) and \t (tab)
- Each line is its own command statement
- Two or more physical lines may be joined into logical lines using backslash characters (\)

0.4.3 Variables

> The purpose of a variable is to store one value. This could be a string, integer, float, Boolean, char, or object.

Variable names

- must start with a letter
- may contain the underscore
- may contain numbers
- case sensitive

The equal sign is used to assign a value to the variable

```
dogname = "spot"
salary = 100000.00
age = 4
```

0.4.4 Order of Operations

- () parenthesis
- ** power
- + - add subtract
- * \ multiple divide
- // integer division
-

0.4.5 Comments

> **Note**
> *Comments* allow the programmer to document their code. **It is very import to document your code.** Comments can be single lined, which is desginated by using a # followed by a space. Or they can be multiple lined in which they are enclosed by three double quotes **“”"** .

0.4.6 Datatypes

```
"""
data types
storing variables
    variable names should be descriptive
"""

#integers
a = 42
print(a)
print(type(a))

# = floats
b = 42.0
print(b)
print(type(b))

# = strings
c = "forty two"
print(c)
print(type(c))

# = booleans  True / False
d = True
print(d)
print(type(d))

# = null
e = None
print(e)
print(type(e))

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#   containers - discuss more in future - just show
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

print("\n*** Lists ***\n")      # Explain \n and \t

### LISTS ###
## properties: ordered, iterable, mutable, can contain multiple data types
mylista = [1,2,3,4]
print(mylista)

mylistb = ["larry", "curly", "moe"]
print(mylistb)
```

```
mylistc = [42, 42.0, "forty two", True, ['Tom', 'Dick', 'Harry']]
print(mylistc)

### TUPLES ###
## properties: ordered, iterable, immutable, can contain multiple data types
## like lists, but they don't change size
mytuplea = (1,2,3)
print(mytuplea)

mytupleb = ("Ducks", "Goats", "Cows")
print(mytupleb)

mytuplec = (1, 1.1, "one", False, [42, 43, 44, 45])
print(mytuplec)
print(type(mytuplec))


### DICTIONARIES ###
## properties: unordered, iterable, mutable, can contain multiple data types
## made of key -value pairs
## keys must be unique, and can be strings, numbers, or tuples
## values can be any type

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

print("\n*** Dictionaries ***\n")

### DICTIONARIES ###
## properties: unordered, iterable, mutable, can contain multiple data types
## made of key -value pairs
## keys must be unique, and can be strings, numbers, or tuples
## values can be any type

mydict = {"key1": "value1", "key2": "value2", "key3": "value3" }
print(mydict)

# student number, last name
mydict1 = {111 : "Smith", 222 : "Doe", 333 : "Baker"}
print(mydict1)
print(type(mydict1))

#show error
mydict1 = {111 : "Smith", 222 : "Doe", 333 : "Baker", 111 : "Adam"}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

print("\n*** Sets ***\n")

### SETS ###
## properties: unordered, iterable, mutable, can contain multiple data types
## made of unique elements (strings, numbers, or tuples)
## like dictionaries, but with keys only (no values)
```

```
#like dictionaries, but leave out the values
myset1 = {'python', 'r', 'java', 'c#'}
print(myset1)

myset2 = {1, 2, 3, 4, 5, 6}
print(myset2)
print(type(myset2))

#show error
myset3 = {11,22,33,33, 44,55}
print(myset3)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Example Showing Mathematical Operations

```
print("Get and display input " )
dogname = input("Enter a dog name ")
dogage = float(input("Enter the dog age in years  "))

# numbers must be in a str() function
print("Dog name = " +  dogname + "  age = " + str(dogage))
print("\n\n")



print(" - - operations review - -")
print("Parenthesis 2 + 5 * 7 vs (2 + 5) * 7")
x = 2 + 5 * 7
print(x)
x = (2 + 5) * 7
print(x)

print("3 to the power of 3 3**3")
x = 3**3
print(x)

a = 10.5
b = 2.5


print("Add and subtract 10.5 2.5")
x = a + b
print(x)

x = a - b
print(x)

print("Multiple and divide 10.5 2.5")

x = a * b
print(x)
```

0.5 Basics - Getting Started

0.5.1 Concepts

- Scripted language - does not compile to machine code note: there are separate programs that will compile it into machine code
- Case sensitive
- Space-based - tabbing has meaning

0.5.2 Expanded Comments

Triple quotes part of the program's documentation. They allow you to have multi-lined comments

```
"""
This is comment line 1
This is comment line 2
This is comment line 3
etc
"""
print("Comments are an essential part of programming")
```

0.5.3 Editors

- Idle
- Spyder
- Pycharm
- others....

Run in interactive mode

0.5.4 Coding Format

List appropriate use of white space and tabbing Do not go more than 80 characters Escape characters \n and \t

Two or more physical lines may be joined into logical lines using backslash characters ()

0.5.5 Variable names

- must start with a letter
- may contain the underscore
- may contain numbers
- case sensitive

0.5.6 Lecture Code

```
"""
Concepts
* Scripted language - does not compile to machine code
    note: there are sperate programes that will compile it into machine code
* Case sensitive
* Space based - tabbing has meaning

Explain comments
    triple quotes part of program's documenation system

Editors
    Idle
```

```
    Spyder
    pycharm
    others....
Run in interactive mode

List appropriate use of white space and tabbing
Do not go more than 80 characters
Escape characters \n and \t

Two or more physical lines may be joined
into logical lines using backslash characters (\)

variable names
must start with a letter
may contain the underscore
may contain numbers
case sensitive

input

Discuss operations
() parenthesis
** power
+ - add subtract
* / multiple divide
//   integer division
% mod


"""
# first program
print ("hello world")

"""
data types
storing variables
    variable names should be descriptive
"""

#integers
a = 42
print(a)
print(type(a))

# = floats
b = 42.0
print(b)
print(type(b))

# = strings
c = "forty two"
print(c)
print(type(c))
```

```
# = booleans  True / False
d = True
print(d)
print(type(d))

# = null
e = None
print(e)
print(type(e))

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#   containers - discuss more in future - just show
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

print("\n*** Lists ***\n")      # Explain \n and \t

### LISTS ###
## properties: ordered, iterable, mutable, can contain multiple data types
mylista = [1,2,3,4]
print(mylista)

mylistb = ["larry", "curly", "moe"]
print(mylistb)

mylistc = [42, 42.0, "forty two", True, ['Tom', 'Dick', 'Harry']]
print(mylistc)

### TUPLES ###
## properties: ordered, iterable, immutable, can contain multiple data types
## like lists, but they don't change size
mytuplea = (1,2,3)
print(mytuplea)

mytupleb = ("Ducks", "Goats", "Cows")
print(mytupleb)

mytuplec = (1, 1.1, "one", False, [42, 43, 44, 45])
print(mytuplec)
print(type(mytuplec))


### DICTIONARIES ###
## properties: unordered, iterable, mutable, can contain multiple data types
## made of key -value pairs
## keys must be unique, and can be strings, numbers, or tuples
## values can be any type

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

print("\n*** Dictionaries ***\n")

### DICTIONARIES ###
```

```
## properties: unordered, iterable, mutable, can contain multiple data types
## made of key -value pairs
## keys must be unique, and can be strings, numbers, or tuples
## values can be any type

mydict = {"key1": "value1", "key2": "value2", "key3": "value3" }
print(mydict)

# student number, last name
mydict1 = {111 : "Smith", 222 : "Doe", 333 : "Baker"}
print(mydict1)
print(type(mydict1))

#show error
mydict1 = {111 : "Smith", 222 : "Doe", 333 : "Baker", 111 : "Adam"}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

print("\n*** Sets ***\n")

### SETS ###
## properties: unordered, iterable, mutable, can contain multiple data types
## made of unique elements (strings, numbers, or tuples)
## like dictionaries, but with keys only (no values)

#like dictionaries, but leave of the values
myset1 = {'python', 'r', 'java', 'c#'}
print(myset1)

myset2 = {1, 2, 3, 4, 5, 6}
print(myset2)
print(type(myset2))

#show error
myset3 = {11,22,33,33, 44,55}
print(myset3)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

print("Get and display input " )
dogname = input("Enter a dog name ")
dogage = float(input("Enter the dog age in years  "))

# numbers must be in a str() function
print("Dog name = " +  dogname + "  age = " + str(dogage))
print("\n\n")



print(" - - operations review - -")
print("Parenthesis 2 + 5 * 7 vs (2 + 5) * 7")
x = 2 + 5 * 7
print(x)
```

```
x = (2 + 5) * 7
print(x)

print("3 to the power of 3 3**3")
x = 3**3
print(x)

a = 10.5
b = 2.5


print("Add and subtract 10.5 2.5")
x = a + b
print(x)

x = a - b
print(x)

print("Multiple and divide 10.5 2.5")

x = a * b
print(x)


# End of program
```

End of Topic

0.6 Second Program

```
"""
Programmer: James Goudy
Date:
Assignment:

2nd program

Problem 1 : Calculate the circumference of a circle
Formula:    C = 2 \piR
            C - circumference
            \pi - pi 3.14
            R - Radius
            D - Diameter

Problem 2 : Calculate the volume of a prism
Formula:    V = lwd
            V - Volume
            l - length
            w - width
            d - depth

Exit Message

DO NOT SINGLE SPACE USE THE WHITE SPACE APPROPRIATELY

Pattern
Ask a question / get data
manipulate the data
Give back the answer

 """


""" Circumference of a circle """
answer = 0.0
r = 0.0
#note: their are not constants in python - designate in all caps
PI = 3.1416

print("\n\nCalculate the circumference of a circle")

r = float(input("Enter the radius  "))

answer = 2.0 * r * PI
answer = round(answer, 2)

print("The circumference of the circle is " + str(answer))

""" Volume of a prism """
print("\n - - - - - - - - -\nCalculate the volume of a prism")

l = float(input("Enter the length "))
```

```
w = float(input("Enter the width "))
d = float(input("Enter the depth  "))

answer = l * w * d
answer = round(answer, 2)

print("The volume of the prism is " + str(answer))

# Exit
print("\nThank you for using Goudy Software")
```

End of Topic

0.7 Functions

Definition - Function
A **Function** is a segment of code that can be "called" upon and run repeatedly as required. A function has a name, can take optional inputs, and may return data to whichever function called it.

A function is a block of organized, reusable code that is used to perform a specific task. It helps to break down a complex problem into smaller, manageable functions.

0.7.1 Structure of a Function

```
def function_name(parameters):
  """Docstring: Explains what the function does"""
  # Function body
  return value  # Optional
```

Code:

- `def`: Keyword to define a function.
- `function_name`: The name you give to the function.
- `parameters`: Optional input values for the function.
- `docstring`: Optional string explaining the function's purpose.
- `return`: Optional keyword to return a value from the function.

```
# The code pattern of a function
def functionName(input):
    # function code here
    # function code here
    # function code goes here

    # return is optional depending the
    # requirements of the function
    return somedata
```

0.7.2 Function Names

Function names follow the same rules as variable names:

- must start with a letter
- may contain the underscore
- may contain numbers
- case sensitive
- no spaces in the name

Special Function - Main
main() This is a special function. Most languages have a function called main(). This is where a program will start and end. While it is not required in python, it is good form and practice to use a **main** function in python.

0.7.3 Program Structure

```
def function1(input):
    # function code here
    # ...
    # function code goes here
    return somedata

def function2(input):
    # function code here
    # ...
    # function code goes here

def function3():
    # function code here
    # ...
    # function code goes here
    return somedata

def function4():
    # function code here
    # ...
    # function code goes here

def main():

    # variables
    x = 0
    ans = 0;

    # assume that the functions 1 and 3
    # will return an integer

    ans = function1(x)

    function2(x)

    ans= function3()

    function4()


# program will start here
main()
```

Can you put the functions below the main? The answer is yes. However, if you code between languages, you may code in a language where it uses a one-pass compiler. This means the compiler will run once from the top to the bottom of the code. So any function calls, the functions need to be "introduced" first before they can be called. Placing the functions at the top will introduce to the compiler the functions before they are called. So if you code between a lot of languages, you might want to adopt a style that will work for the majority of languages.

Definition - Compiler

Compiler a program that converts your source code into machine-code form (ones and zeros) so that they can be read and executed by a computer.

0.7.4 Lecture Code

Example 1

```
"""
Function Demo
"""
# Add two numbers - Functions requires 2 numbers
# and returns the total
def add2nums_a(num1, num2):
    ans = 0
    ans = num1 + num2
    return ans

# add two numbers B
# Add two numbers - Functions requires 2 numbers
# returns nothing, and displays the answer
def add2nums_b(num1, num2):
    ans = 0
    ans = num1 + num2
    print("The answer is " + str(ans))

# add two numbers C
# Add two numbers - the asks for the numbers
# and returns the answer
def add2nums_c():
    ans = 0
    num1 = float(input("Enter first number " ))
    num2 = float(input("Enter second  number " ))
    ans = num1 + num2
    return ans

# add two numbers d
# Add two numbers - the function does everything
# Function asks for the numbers and displays the answer
def add2nums_d():
    ans = 0
    num1 = float(input("Enter first number " ))
    num2 = float(input("Enter second  number " ))
    ans = num1 + num2
    print("The answer is " + str(ans))


# - - - - - - - - - - - - - - - - InClass Dogyears - - - - - - - - - - - - - - - - -
def dogYears_a(humanYears):
    ans = 0
    ans = humanYears * 7
    return ans

def dogYears_d():
```

```
    ans = 0.0
    humanYears = 0.0

    humanYears = float(input(" Please enter human years  "))

    ans = humanYears * 7

    print("DogYears D - Dogs years is equal to " + str(ans))


# - - - - - - - - - - - - - Greetings A - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def Greetings_A( firstName,  lastName,  PlayerNum):

    ans = ""

    ans = "Welcome " + firstName + " " + lastName + " player " + str(PlayerNum)

    return ans

def main():

    #variables
    xNum1 = 0
    xNum2 = 0
    xAns = 0

    xhumanYears = 0

    xFirstName = ""
    xLastName = ""
    xAnsString = ""
    xPlayerNumber = 0


    # Base Code
    print("Add two numbers");
    xnum1 = float(input("Please enter first number  "))

    xnum2 = float(input("Please enter second number  "))

    xAns = xNum1 + xNum2

    print("The answer is " + str(xAns))

    # Example A
    print("\n\n - - - - - - - - - Example A - - - - - - - - -\n\n")

    xnum1 = float(input("A Please enter first number  "))

    xnum2 = float(input("A Please enter second number  "))

    xAns = add2nums_a(xnum1, xnum2)
```

```
print("A The answer for add2nums_A is " + str(xAns))

# Example B
print("\n\n - - - - - - - - - Example B - - - - - - - - -\n\n")

xnum1 = float(input("B Please enter first number  "))

xnum2 = float(input("B Please enter second number  "))

add2nums_b(xnum1, xnum2)







# EXample C
print("\n\n - - - - - - - - - Example c - - - - - - - - -\n\n")

xAns = add2nums_c()

print("C The answer for add2nums_c is " + str(xAns))

# Example D
print("\n\n - - - - - - - - - Example d - - - - - - - - -\n\n")

add2nums_d();

# Inclass - Dog Years
print("\n\n - - - - - - - - - Example dogyears A - - - - - - - - -\n\n")


xhumanYears = float(input("DogYears A Please enter human years  "))

xAns = dogYears_a(xhumanYears)

print("DogYears A Dogyears " + str(xAns))

print("\n\n - - - - - - - - - Example dogyears D - - - - - - - - -\n\n")

dogYears_d()

# - - - - - - - - - - - - - - - - - - Greeting_A - - - - - - - - - - - - - - - - - - - - - -

print("\n\nGreeting")
xFirstName= input("Please enter your first name  ")


xLastName = input("Please enter your last name  ")

xPlayerNumber = int(input("Please enter your player number  "))
```

```
    xAnsString = Greetings_A(xFirstName, xLastName, xPlayerNumber)

    print(xAnsString)

    # - - - - - - - - - - - - - - - End Of Program - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    print("Thank you for using Goudy Software - good bye")



# The function where a program starts and ends
main()
```

Example 2

```
"""
Function Demo 2

@author: jgoudy
"""

import math

# calcuate the volume of a cylinder
# pR^2h
def cylinderVolume(radius, height):

    ans = 0.0

    ans = math.pi * radius * radius * height

    ans = round(ans, 2)

    return ans

    # - - - - - - - - - -   End of Volume of Cylinder - - - - - - - - - - -

# calculate the volume of a cone
# v = p r^2(h/3.0)
def coneVolume():

    # variables
    r = 0.0
    h = 0.0
    ans = 0.0

    print("\nCalculate Volume of a Cone")

    # get data
    r = float(input("Enter radius of cone: "))
    h = float(input("Enter height of cone: "))

    ans = math.pi * r * r *(h / 3.0)
    ans = round(ans, 2)
```

```
    print("The volumen the cone is " + str(ans))

    # - - - - - - - End of Volume of Cone - - - - - - - - - - - - - - -

def main():

    # variables
    r = 0.0
    h = 0.0
    ans = 0.0

    print("\nCalculate Volume of a Cylinder")

    # get data
    r = float(input("Enter radius of cylinder: "))
    h = float(input("Enter height of cylinder: "))

    # call the function
    ans = cylinderVolume(r,h)

    print("The volume of a cylinder is " + str(ans))

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    coneVolume()


    # - - - - - - - End of main - - - - - - - - - - - - - - - - -

#program starts here
main()
```

0.7.5 Exercise

Program the following:

```
"""
volumeRecPrism  - (1enght, width, height) has a return
rec/prism volume = (l * w) * h
(do like cylinder)

volumeTriPrism - ()  no return
triangle/prism =  1.0/2.0 *(l * w) * h
(do like cone)
"""
```

End of topic

0.8 Debugger

A debugger is a computer program used to test and identify errors (bugs) in other programs. It acts as a troubleshooting tool for programmers by providing a controlled environment to examine a program's execution and pinpoint malfunctions.

Here's a breakdown of its key functionalities:

- **Execution Control:** Debuggers allow you to pause the program's execution at specific points, typically called breakpoints. This enables you to inspect the program's state at that moment.
- **Variable Inspection:** Debuggers provide ways to examine the values of variables at breakpoints. This helps you understand how data is flowing through your program and identify potential issues with calculations or assignments.
- **Stepping Through Code:** Debuggers offer features like single-stepping, where you can execute the program one line of code at a time. This allows you to follow the program's logic step by step and see how it interacts with variables.
- **Stack Trace Analysis:** When an error occurs, debuggers often display a stack trace. This information reveals the chain of function calls that led to the error, making it easier to pinpoint the problematic section of code.

In essence, a debugger acts as a bridge between the programmer and the running program. By offering insights into the program's internal state and execution flow, it empowers developers to effectively identify, understand, and fix bugs, ultimately leading to more reliable and efficient software

0.8.1 Demo Code

```
'''
Programmer: James Goudy
'''
def add(num1, num2):
  """Adds two numbers and returns the sum."""
  breakpoint()  # Add breakpoint before the calculation
  return num1 + num2

def subtract(num1, num2):
  """Subtracts two numbers and returns the difference."""
  breakpoint()  # Add breakpoint before the calculation
  return num1 - num2

def loop_function():
  """Runs a loop for 5 times, printing a message each time."""
  breakpoint()  # Add breakpoint before the loop
  for i in range(5):
    print(f"Loop iteration: {i+1}")

def main():
  """Main function to call other functions and start the program."""
  print("Welcome to the calculator program!")

  # Get user input for numbers
  number1 = float(input("Enter the first number: "))
  number2 = float(input("Enter the second number: "))

  # Call add and subtract functions
  sum = add(number1, number2)
  difference = subtract(number1, number2)
```

```
  # Print the results
  print(f"The sum of {number1} and {number2} is: {sum}")
  print(f"The difference of {number1} and {number2} is: {difference}")

  # Call the loop function
  loop_function()

# Call the main function to start the program
if __name__ == "__main__":
  main()
```

This program defines four functions:

- `add`: Takes two numbers as arguments and returns their sum.
- `subtract`: Takes two numbers as arguments and returns their difference.
- `loop_function`: Runs a loop for 5 times, printing the current iteration number.
- `main`: The main function that welcomes the user, gets input for numbers, calls the `add` and `subtract` functions with the user input, prints the results, and then calls the `loop_function`.

The `if __name__ == "__main__":` block ensures that the `main` function only runs when the script is executed directly, not when imported as a module.

0.8.2 Commands

The built-in Python debugger, `pdb`, offers a variety of commands to control program execution and inspect variables during debugging. Here's a breakdown of some essential commands:

Setting Breakpoints:

- `b (break)`: Sets a breakpoint at a specific line number or function name. You can optionally specify a condition for the breakpoint to trigger only when the condition evaluates to `True`.

Navigating Code Execution:

- `c (continue)`: Continues program execution until the next breakpoint is hit.
- `s (step)`: Steps into the next line of code, including function calls.
- `n (next)`: Steps over the next line of code, without entering functions.

Examining Values:

- `p (print)`: Prints the value of an expression in the current context.

Stack Management:

- `u (up)`: Moves the debugger up one level in the call stack (useful for inspecting function arguments).
- `d (down)`: Moves the debugger down one level in the call stack (helpful for following function calls).

Other Useful Commands:

- `l (list)`: Lists the source code surrounding the current line (default: 11 lines).
- `q (quit)`: Exits the debugger.
- `h (help)`: Provides help on available debugger commands.

Starting the Debugger:

There are two main ways to launch the debugger:

1. **Using `python -m pdb your_script.py`**: This runs your script (`your_script.py`) under the control of `pdb`.
2. **Using `breakpoint()`**: Inside your code, add `breakpoint()` at the desired breakpoint location. *This automatically launches `pdb` when the program execution reaches that line (available in Python 3.7 and later).*

For a comprehensive list of commands and their functionalities, refer to the official `pdb` documentation: https://docs.python.org/3/library/pdb.html

0.9 Logging

0.9.1 Python Logging and Log Files

Python logging provides a structured way to record events and messages within your application. These messages are typically written to log files, but can also be directed to other destinations. Here's a breakdown of the concepts:

Logging:

- **Purpose:** Logging allows you to track the execution of your program, pinpoint errors, analyze application behavior, and gain insights into user interactions.
- Benefits:
  - **Debugging:** Logs provide a detailed trail of what happened during program execution, aiding in identifying the source of errors.
  - **Monitoring:** Logs help you understand how your application behaves under different conditions and identify potential issues before they become critical.
  - **Auditing:** Logs can be used to track user activity and maintain an audit trail for security purposes.

Log Files:

- **Destination:** Log messages are typically written to text files, although they can also be sent to network sockets, email, or other destinations.
- Content:
  Log files contain information about the logged messages,
  including:
  - **Level:** The severity level of the message (debug, info, warning, error, critical).
  - **Timestamp:** The date and time the message was logged.
  - **Logger Name:** The name of the module or component that generated the message.
  - **Message:** The actual content of the message you logged.
- Benefits:
  - **Centralized Monitoring:** Logs provide a centralized location to track application activity from a single source.
  - **Persistence:** Logged information persists even after the program terminates, allowing for historical analysis.
  - **Customization:** You can configure logging to capture specific types of information relevant to your needs.

More information at: https://docs.python.org/3/howto/logging.html

0.9.2 Logging Levels

The logging levels in Python's `logging` module define the severity of a message. They are used to categorize log messages and control which ones get recorded based on their importance. Here's a breakdown of the levels and their typical uses:

- **DEBUG (10):** These are detailed, low-level messages intended for diagnosing issues during development. They might include variable values, function calls, or internal program flow. You'd typically only enable DEBUG logging when actively debugging a specific problem.
- **INFO (20):** Informational messages that confirm expected behavior. These are useful for tracking the general progress of your application and monitoring its state. Examples include successful logins, database connections established, or configuration settings loaded.
- **WARNING (30):** These indicate potential problems or unexpected conditions. WARNING messages don't necessarily mean there's a critical issue, but they deserve attention. Examples include low disk space, invalid user input, or resource limitations.
- **ERROR (40):** These indicate serious errors that may have prevented the application from performing some functionality. They typically require intervention to fix. Examples include failing to connect to a database, encountering unexpected exceptions, or data validation errors.

- **CRITICAL (50):** Critical messages indicate a severe error that likely caused the application to crash or become unusable. These are high-priority issues requiring immediate attention. Examples include critical system failures, data corruption, or security breaches.

Here's a general guideline for choosing the appropriate level:

- Use DEBUG for detailed troubleshooting information.
- Use INFO for normal application events.
- Use WARNING for potential issues that may become problems later.
- Use ERROR for serious errors that prevent functionality.
- Use CRITICAL for system crashes or unrecoverable situations.

By effectively using logging levels, you can create a log file that is informative and helps you identify and fix problems efficiently.

0.9.3 Run Mode

In Python, there's no specific "run mode" required to create log files using the `logging` module. Your program can generate log files as long as the logging configuration is set up and the program runs to execute the logging statements.

Here's why:

- The `logging` module operates within your Python program itself. It doesn't rely on external factors like a development environment or specific execution mode.
- As long as your program reaches the lines of code where you call logging methods (like `debug`, `info`, etc.), corresponding log messages will be generated based on your configuration.
- This configuration typically happens using `basicConfig` or a more advanced logging configuration file. This setup defines where the logs go (often a file) and the level of detail captured (debug, info, etc.).

Therefore, you can create log files regardless of whether you're running your program from the command line, within an IDE, or even embedded within another application. As long as the program executes the logging statements, logs will be generated based on your configuration.

> **Note**
> VSCode seems to be the exception. The Log file will only create when ran from the debug mode.

0.9.4 Lecture Code

```
'''
Programmer: James Goudy
Logging lecture code

NOTE: for the log file to run in VSCODE,
 the program needs to run in logging mode

If the program runs from the command prompt,
it will create the log files
'''



import logging
logger = logging.getLogger(__name__)

def setupLogging():
    # setup logging file with force=True to overwrite existing handlers
    logging.basicConfig(filename='myLogfile.log',
```

```
                        level=logging.DEBUG,
                        format='%(asctime)s:%(levelname)s:%(message)s',
                        force=True)


def levelMessages():

    # demo logging messages
    logger.debug("A debug message")
    logger.info("An info message")
    logger.warning("An warning message")
    logger.error("An error message")
    logger.critical("A critical message")


def example1():

    num1 = 10
    num2 = 0
    ans = 0

    try:
        ans = num1 / num2
    except Exception as err:
        logger.critical(err)

def example2():

    for i in range(10):
        logger.info("i = " + str(i))

def main():
    setupLogging()

    levelMessages()

    example1()

    example2()

    print("bye")

main()

'''
Output:
2024 -07 -23 18:42:39,485:DEBUG:A debug message
2024 -07 -23 18:42:39,487:INFO:An info message
2024 -07 -23 18:42:39,487:WARNING:An warning message
2024 -07 -23 18:42:39,488:ERROR:An error message
2024 -07 -23 18:42:39,488:CRITICAL:A critical message
2024 -07 -23 18:42:39,488:CRITICAL:division by zero
2024 -07 -23 18:42:39,488:INFO:i = 0
```

```
2024 -07 -23 18:42:39,488:INFO:i = 1
2024 -07 -23 18:42:39,488:INFO:i = 2
2024 -07 -23 18:42:39,489:INFO:i = 3
2024 -07 -23 18:42:39,489:INFO:i = 4
2024 -07 -23 18:42:39,489:INFO:i = 5
2024 -07 -23 18:42:39,489:INFO:i = 6
2024 -07 -23 18:42:39,489:INFO:i = 7
2024 -07 -23 18:42:39,489:INFO:i = 8
2024 -07 -23 18:42:39,490:INFO:i = 9
'''
```

0.10 Decision Trees

0.10.1 Making Decisions in Your Code

If statements are fundamental to programming logic. They allow your code to make decisions based on specific conditions. If a condition is true, a block of code is executed; otherwise, it is skipped.

```
"""

if statements

NOTE: There are no switch statements in Python
as of version 3

"""

def if_example_one(num):

    print("\nExample 1")

    # Check for True only
    if(num < 100):
        # run this code if true
        print("Your number is less than 100")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def if_example_two(num):


    print("\nExample 2")
    # Check for true, else the code will do code if it's false
    if(num < 100):
        # run this if true
        print("Your number is less than 100")
    else:
        # run this code if false
        print("Your number is greater than 99")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def if_example_three(anum):

    print("\nExample 3")

    #check for mutliple cases of true
    if(anum < 10):
        # run this if true
        print(str(anum) + " is less than 10")
    elif(anum < 20):
        # run this if true
        print(str(anum) + " is less than 20")
    elif(anum < 30):
        # run this if true
        print(str(anum) + " is less than 30")
```

```
    else:
        # run this if false
        # the else optional - and not required.
        print("Your number is greater than 29")
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def if_range(salary):

    print("\nExample Check a number between a range")
    # Check to see if a number is between a range
    if(salary > 0 and salary <= 50000):
        print("\nNeed to win the lottery")
    elif(salary > 50000 and salary <= 200000):
        print("\nMiddle Class")
    elif(salary > 200000 and salary <= 10000000):
        print("\nTop Ten")
    else:
        print("\nLiving On Easy Street")

    # Note: if(salary > 0  <= 50000):
    # Python can write an expression this way
    # However - this is NOT accepted in most other
    # programming languages.

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def if_strings(word1, word2):

    print("\nCompare Strings")

    # Display the words
    print("Word1 = " + word1 + " | Word2 = " + word2)

    # Check if one word is equal to another  i.e. dog equals dog
    # Text is case sensitive - i.e.  Dog does not equal dog
    if(word1 == word2):
        print(word1 + " equals " + word2)
    else:
        print(word1 + " does not equals " + word2)

    # other features of python (ONLY)!!!!!!!!!!!!!!!!!!!!!!!!
    # Use Case - if you had a list of words that were of same case
    # meaning they were ALL upper case or lower case
    # you can use the technics below to see if a word
    # came before or after another word.
    if(word1 > word2):
        print(word1 + " greater than " + word2)

    if(word1 >= word2):
        print(word1 + " greater than or equal to " + word2)

    if(word1 < word2):
        print(word1 + " less than " + word2)
```

```
    if(word1 <= word2):
        print(word1 + " less than or equal to " + word2)

    if(word1 != word2):
        print(word1 + " not equal to " + word2)


# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -


def main():


    anum = float(input("Please enter a number:  "))

    if_example_one(anum)

    if_example_two(anum)

    if_example_three(anum)

    mySalary = float(input("Please enter your salary:  "))

    if_range(mySalary)

    w1 = input("Enter Word 1 ")
    w2 = input("Enter Word 2 ")

    if_strings(w1,w2)


# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

main()
```

0.11 How To Create A Menu

```
# -* - coding: utf -8 -* -
"""
Menu Template
Created on Tue Sep 17 17:06:05 2019

@author: jgoudy
"""

def Function1():
    print("This is function 1")

def Function2():
    print("This is function 2")

def Function3():
    print("This is function 3")
    WinAPrize()

def WinAPrize():
    print("You have won 5000 live grasshoppers - Enjoy")

def Menu():
    print("\nMenu Example\n" + \
          "1. Function 1\n" + \
          "2. Function 2\n" + \
          "3. Function 3" )

    choice = int(input("Choose a number:  "))

    if(choice == 1):
        Function1()
    elif(choice == 2):
        Function2()
    elif(choice == 3):
        Function3()
    else:
        print("That was not a choice")


def main():
    Menu()

    print("\nbye bye")


main()

,,,
Program choices
1. Happy Function 1
2. Happy Function 2
3. Happy Function 3
```

```
Please choose 1, 2, or 3:
'''
```

0.12 Nested Ifs

0.12.1 Lecture Code

Example I - Should I Post This

```
# -* - coding: utf -8 -* -
"""
Program - Should I post this?

Created on Fri Sep 20 11:49:44 2019

@author: jgoudy

NOTE: lower converts input into lower case

Cases have to match when comparing for true
"""

def PostPhoto():

    choice = input("Is everyone clothed? y/n  ").lower()

    if(choice == "y"):
        #person is fully clothed
        #ask if doing anything embarrassing
        choice = input("are you doing anything "  \
                       + "illegal or embarrassing? y/n  ").lower()

        if(choice == "y"):
            print("Stop - do not post")
        elif(choice == "n"):
            print("Ok to post")
        else:
            print("That was not a choice")


    elif(choice == "n"):
       # Person is not fully clothed
       print("Stop - do not post")
    else:
       print("That was not a choice")


def PostText():

    choice = input("Are there expletives or abrasive content? y/n  ").lower()

    if(choice == "y"):
       # Text has objectionable content
       print("Stop - do not post")

    elif(choice == "n"):
        # Text has no objectionable content
        #ask if they were partying
```

```
        choice = input("Have you been partying?  y/n  ").lower()

        if(choice == "y"):
            print("Stop - do not post")
        elif(choice == "n"):
            print("Ok to post")
        else:
            print("That was not a choice")



def ShouldIPost():

    print("Should I post this?")
    choice = int(input("Press 1 for text and 2 for photo: "))

    if(choice == 1):
        PostText()
    elif(choice == 2):
        PostPhoto()
    else:
        print("That was not a choice")


def main():
    ShouldIPost()


main()
```

Example 2 - Get Out Of Bed

```
# -* - coding: utf -8 -* -
"""
Spyder Editor
By James Goudy


The program assumes that if the user picks anything but a y
the answer will be no

Remember .lower() converts the input to lowercase
"""
import random

# ui = variable name short for user input
def weekend():

    #
    ui = input("Is it the weekend y/n ").lower()

    if(ui == "y"):

        ui = input("Own a dog  y/n ").lower()
```

```
        if(ui == "y"):
            print("Get out of bed and let the dog out")
        else:
            print("Stay in bed")
    else:
        vacation()

def vacation():

    ui = input("Are you on vacation y/n ").lower()

    if(ui == "y"):
        print("Stay in bed")
    else:
        job()




def job():

    ui = input("Do you have a job y/n ").lower()

    if(ui == "n"):
        print("Stay in bed")
    else:
        print("Get out of bed.\nPrepare for work.")
        leaveforwork()

def leaveforwork():
    # We will generate a random number
    # and let that determine if we have to
    # grab the keys or the transit card
    # will will gamify this by not telling
    # the user what the computer has chosen
    # the user will have to guess


    # Generate a random integer
    aRandomInteger = random.randint(0,10)

    # % is the modulus operator
    # in integer division it returns the remainder
    # note we are dividing by 2 and will return
    # the remainder. Since we are dividing by two
    # if the random integer is even, the remainder
    # will be 0

    carstatus = (aRandomInteger % 2)

    print("Pick up either the car keys or transit pass")
    ui = int(input("press 1 for keys or 2 for transit pass"))
```

```
    # We will make the following assumption
    # if carstatus equals 0
    #    then the car is gone and
    #    the transit pass is needed to go to work
    # else cartstatus does not equal 0
    #    then the car is here
    #    the keys are needed to go to work

    if(carstatus == 0):
        # this means that the car is gone
        # person needs the transit pass
        print("The car is gone")
        if(ui == 2):
            print("You picked up the transit pass")
            print("and you are off to work")
        else:
            print("You picked up the keys")
            print("\nGo back choose again")
            leaveforwork()
    else:
        # this means that the car is here
        # person needs the keys
        print("The car is here")
        if(ui == 1):
            print("You picked up the keys")
            print("and you are off to work")
        else:
            print("You picked up the transit pass")
            print("\nGo back choose again")
            leaveforwork()




def main():
     print("Get Out Of Bed Program")

     status = "y"
     while(status == "y"):
         weekend()

         status = input("would you like " \
                        +"to run the program again y/n ").lower()

     print("\nBye bye enjoy your day")

main()
```

0.13 If ___name__ == ___main___

In Python, the `if __name__ == '__main__':` construct is used to control the execution of code based on whether the script is being run directly or being imported as a module.

How it works:

- `__name__`:
  This is a special variable in Python that gets assigned a value depending on how the script is being used.
  - If the script is being run directly, `__name__` is set to `'__main__'`.
  - If the script is being imported as a module, `__name__` is set to the name of the module.
- `if __name__ == '__main__'`::
  This conditional statement checks if the `__name__` variable is equal to `'__main__'`. If it is, the code within the indented block will be executed.

Why is it useful?

- Separating executable code from module code:
  This construct allows you to include code that should only be executed when the script is run directly, while keeping other code (functions, classes, etc.) available for use when the script is imported as a module.
- Organizing code:
  It helps to structure your code in a way that separates the main execution logic from the reusable components.
- Testing:
  It is a common practice to place unit tests within an `if __name__ == '__main__':` block, so they are only executed when the script is run directly, and not when it's imported.

0.13.1 Example:

NOTE: both files need to be in the same directory

FILE ONE "add2.py"

When add2 is ran by itself, only the explanation() function runs

```
# if __name__ lecture code
# Developer: James Goudy

def add2Nums(num1,num2):
    try:
        num1 = float(num1)
        num2 = float(num2)

        return (num1 + num2)

    except Exception as e:

        print(f"\nERROR: {e}\n")

        return None

def explain():
    msg = '''
    A python script can be ran by itself.
    Or it can be imported as a module in
    another script.
```

```
    If they script is ran by itself,
    then the python special variable __name__
    will automatically be named "__main__".

    When it is "main", then other code
    underneath that if statement will run.

    '''
    print("\n")
    print(__name__)
    print(msg)

# this acts like  main()
if __name__ == '__main__':

    print(f"\nThis is add2.py")

    # The following will only run
    # if the code is ran separately
    # and not imported into another
    # file
    explain()

    print("\n\nbye\n")

# - - - - - - - - - - - Output - - - - - - - - - - -
'''

This is add2.py

__main__

    A python script can be ran by itself.
    Or it can be imported as a module in
    another script.

    If they script is ran by itself,
    then the python special variable __name__
    will automatically be named "__main_".

    When it is "main", then other code
    underneath that if statement will run.


bye
'''
```

FILE TWO "testadd2.py"

```
# Developer: Jame Goudy
# add2.py file is imported

import add2
```

```
def addMyNums():
    print("\nAdd 2 numbers")
    n1 = input("Enter 1st number: ")
    n2 = input("Enter 2nd number: ")

    # add2.add2Nums is imported
    ans = add2.add2Nums(n1,n2)

    print(f"Sum = : {ans}" )


# this acts like  main()
if __name__=='__main__':

    # NOTE: explanation does not run
    # from add2.py

    print(f"\nThis is testadd2.py")
    print(__name__)

    addMyNums()

    print("\n\nbye\n")

# - - - - - - - - - - - Output - - - - - - - - - - -
'''
This is testadd2.py
__main__

Add 2 numbers
Enter 1st number: 20
Enter 2nd number: 80
Sum = : 100.0


bye
'''
```

In summary:

- The `if __name__ == '__main__':` construct is a valuable tool for controlling code execution and creating modular, reusable Python scripts.
- It is widely used in Python development and is considered a best practice for writing well-structured code.

(JGGEM)

0.14 Try Except - Preventing Errors

0.14.1 Key Ideas

- try
- except
- finally

The goal of writing code is to prevent errors from crashing the program. A try-and-except statement will catch errors and prevent the program from crashing.

Attention
Any code that can possibly cause the code to fail should be placed in a try-except statement. Note other languages call this try and catch.

0.14.2 Example Code

```
"""
Created on Thu Sep 22 14:57:31 2022

@author: jgoudy

"""

def example():

    x = "Bob"
    y = 10.0

    print("Error 1 Example")
    try:
        # print the 101 character
        # this will fail because "Bob" only has three characters
        print(x[100])
    except Exception as e:
        print(e)


    print("\nError 2 Example\n")
    try:
        # This will fail because you cannot divide by zero
        # the code under finally will always run
        t = y / 0
        print(t)
    except Exception as e:
        print("*** Error ***")
        print(e)
    finally:
        print("finally code - This code will always run")
```

```
    print("\nError 3 Example\n")
    try:
        # note that this code will not fail
        # the code under finally will always run
        t = y / 1
        print(t)
    except Exception as e:
        print("*** Error ***")
        print(e)
    finally:
        print("finally code - This code will always run")



def main():
    example()

    print("\nbye bye")


main()

'''
Output:

Error 1 Example
string index out of range

Error 2 Example

*** Error ***
float division by zero
This code will always run

Error 3 Example

10.0
This code will always run

bye bye

'''
```

0.15 Iteration

Loops in Python are used to execute a block of code repeatedly until a certain condition is met. They are essential for automating tasks and performing repetitive operations efficiently.

0.15.1 Key Ideas

- For Loops
- While Loops
- Loops are for program and flow control

Definition
Iteration - loops - This is the process of having code repeat itself

---

"For" loops are used if you have a set number of times or items to run.
"While" loops are used if you don't have a specific number of times to run. They will run while a condition is true.

> In most other languages there is a loop that will always run once. It may run more than once, but is set to always run at least once. That loop is the "Do" loop. Python does not have this loop.

Key Purposes of Loops

- **Iterating over sequences:** Processing each element in a list, tuple, or string.
- **Accumulating values:** Calculating sums, products, or other aggregations.
- **Repeating actions:** Performing tasks multiple times based on a condition.
- **Searching for elements:** Finding specific values within a dataset.

Common Use Cases

- Processing data from files or databases.
- Creating repetitive patterns or structures.
- Implementing algorithms and mathematical calculations.
- Building interactive programs with user input.

0.15.2 Lecture Code

Example 1

```
import sys

def for_example():

    # c is a counter and 10 is the stop
    # in this example, the counter starts a 0
    # and ends at nine. 10 is the stop or boundary

    # loops 0 through 10 times
```

```
for c in range(10):
    sys.stdout.write(str(c) + " ")
print()

# change starting and ending points
# for the loop
# loops for numbers 50 - 59
for c in range(50,60):
        sys.stdout.write(str(c) + " ")

print("\n - - - - - - - - - - -\n")

a = 40
b = 50
# note the starting and ending points
 # can be variables
for c in range(a,b):
        sys.stdout.write(str(c) + " ")

print("\n - - - - - - - - - - -\n")

# how to loop through a string
for x in "Python":
    if(x == "h"):
        print(" Boom ")

    print(x)

print("\n - - - - - - - - - - -\n")

# using two for statements to create a grid

# grid of stars
numRows = 5
numColumns = 10

# This for loop controls the number of rows
for row in range(numRows):

    # This for loop controls the number or column and
    # one individual row
    for col in range(numColumns):
        sys.stdout.write("* ")
    print()


print("\n - - - - - - - - - - -\n")

# grid showing the coordinate of the row and column
for row in range(numRows):
    for col in range(numColumns):
        sys.stdout.write("("+ str(row) + "," + str(col) + ") ")
    print()
```

```
    print("\n - - - - - - - - - - - -\n")


    # this form of the for loop allows the programmer
    # to set the start - 0, the end 100(actual ends at 99)
    # and the counting interval 10 (called the "step")
    for cc in range(0,100,10):
        sys.stdout.write(str(cc) + " ")

    # to count backwards, set the first number higer
    # than the middle number and then use a negative
    # counter or step

    print("\n******** Count Backwards **********")
    for ccc in range(100, -1, -10):
        sys.stdout.write(str(ccc) + " ")


def while_example():

    print("\n ******** While *********")

    run = True

    i = 0
    # this loop will continue to repeat itself
    # as long as "i" is less than 60
    while i < 60:
        print(i)
        # i = i + 5
        i += 10

    print("\n - - - - - - - - - - - -\n")

    # while statement using a boolean variable
    # this loop will continue to repeate itself
    # as long as "run" is true
    i = 0;
    while run:
        print("Whoopee it works")
        i+=1
        if(i>10):
            run = False


def main():


    quit = "n"

    # this way the only letter that lets the user quit is y
    while(quit != "y"):
```

```
        for_example()

        while_example()

        quit = input("Would you like to quit y / n ").lower()

    print('bye')

main()

'''
Output
0 1 2 3 4 5 6 7 8 9
50 51 52 53 54 55 56 57 58 59
 - - - - - - - - - - -

40 41 42 43 44 45 46 47 48 49
 - - - - - - - - - - -

P
y
t
 Boom
h
o
n

 - - - - - - - - - - -

* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *

 - - - - - - - - - - -

(0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7) (0,8) (0,9)
(1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7) (1,8) (1,9)
(2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7) (2,8) (2,9)
(3,0) (3,1) (3,2) (3,3) (3,4) (3,5) (3,6) (3,7) (3,8) (3,9)
(4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7) (4,8) (4,9)

 - - - - - - - - - - -

0 10 20 30 40 50 60 70 80 90
******** Count Backwards **********
100 90 80 70 60 50 40 30 20 10 0
 ******** While *********
0
10
20
```

```
30
40
50

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

Whoopee it works
Whoopee it works
Whoopee it works
Whoopee it works
Whoopee it works
Whoopee it works
Whoopee it works
Whoopee it works
Whoopee it works
Whoopee it works
Whoopee it works
Would you like to quit y / n
'''
```

Example 2

```
"""
Iteration / Loops
"""
import sys

def for_example():

    for cntr in range(10):
        sys.stdout.write(str(cntr) + " ")
    # 0 1 2 3 4 5 6 7 8 9
    print()

    for cntr in range(50, 60):
        sys.stdout.write(str(cntr) + " ")
    # 50 51 52 53 54 55 56 57 58 59

    print()

    for letter in "Python":
        print(letter)
    print()

    myWord = "Banana"
    for letter in myWord:
        print(letter)

    for x in range(100):
        sys.stdout.write(str(x) + " ")
        if(x == 20):
            break
        # sys.stdout.write(str(x) + " ")
```

```
        # continue - if you stopped a loop,
        # if you are still in the loop


def while_example():
    print("\nWhile Loops")

    cntr = 0
    while cntr < 10:
        sys.stdout.write(str(cntr) + " ")
        # cntr += 1
        cntr = cntr + 1

    print()
    bb = "ba"
    sys.stdout.write(bb)


    cntr = 0
    while cntr < 100000:
        sys.stdout.write(" na ")
        cntr +=1
        if(cntr == 20):
            break


def main():
    for_example()
    while_example()

main()
```

Example 3

```
# -* - coding: utf -8 -* -
"""
Created on Tue Feb 18 17:49:15 2020

@author: jgoudy
"""
import sys


def main():


    aword = "Bubba"

    # len() this prints the lenght of the variable
    # starting at 1 so bubba has a length of 5
    print(len(aword))

    # note the key word "in"
    # aword is made up of 5 chars
```

```
    # c which is the counter will iterate through
    # each object - which in this case is each letter

    # ord gives the ascii decimal number for the char
    # asciitable.com
    for c in aword:
        print(c + " " +  str(ord(c)) )

    print("\n - - - - - - - - - - - -")

    # if we use "in range" to work with a string
    # variable, we can use the counter to represent
    # each location of the string

    #  aword  = | B | u | b | b | a |
    #           - - - - - - - - - - - - - - - - - - - -
    # position  | 0 | 1 | 2 | 3 | 4 |

    # c will start at positon 0 and
    # go up in value by 1 each time it loops

    # to access the specific letter we use the
    # following patter variablename[counterVariable]
    for c in range(len(aword)):
        sys.stdout.write(aword[c] + " ")

    print("\n - - - - - - - - - - - - -")

    # in the following 10 is your starting number (inclusive)
    # 101 is the stop(exclusive)
    # 5 is your countby
    for c in range(10,101,5):
        sys.stdout.write(str(c) + ",")

    print("\n - - - - - - - - - - - - -")

    # chr will print ascii of the integer arguement
    # it is given - in this case it will print out
    # all of the capital letters A - Z
    for c in range(65, 91):
        sys.stdout.write(chr(c))
    print("\n - - - - - - - - - - - - -")


main()
```

Example 4

```
import sys
import os

def for_example():

    # number of times
```

```
    for c  in range(10):
        sys.stdout.write(str(c) + " ")

    print("\n - - - - - - - - - - -\n")

    a = 15
    for c in range(a):
        sys.stdout.write(str(c) + " ")

    print("\n - - - - - - - - - - -\n")

    # count within a range
    for c in range(50,60):
        sys.stdout.write(str(c) + " ")

    print("\n - - - - - - - - - - -\n")

    # iterate through a string
    aWord = "Python"
    for ch in aWord:
        sys.stdout.write(ch + " | " )

    print("\n - - - - - - - - - - -\n")

    # range(start,finish, countby)
    for x in range(0,20,2):
        sys.stdout.write(str(x) + " | " )

    # iterate backwards by 20
    print("\n - - - - - - - - - - -\n")
    for x in range(100,0, -20):
        sys.stdout.write(str(x) + " | " )

    print("\n - - - - - - - - - - -\n")

    # iterate through items in a list
    stooges = ["Larry", "Curly", "Moe", "Shemp","Curly Joe"]
    for aStooge in stooges:
        sys.stdout.write(str(aStooge) + " | " )

    print("\n - - - - - - - - - - -\n")
    # iterate through items in a list by position
    stooges = ["Larry", "Curly", "Moe", "Shemp","Curly Joe"]
    for x in range(len(stooges)):
        print(str(x+1) + ". " + stooges[x] )

def while_example():

    # while statement
    print("While Loops")

    i = 0
```

```
    while(i < 10):
        print("Python")

        # Note a counter is needed
        # i = i + 1
        i += 2

    # note how a boolean is used in the parentheses

    y = True
    i = 0
    while(y):
        print("bob ross for happy little loops")
        i = i + 1
        if(i >= 12):
             y = False


def try_example():

    # example of try and except

    x = "python"

    try:
        print(x[20])
    except:
        print("you had a error")


def grid_example():

    # grid example
    print("\n - - - - - - - - grid example - - - - - - - - - -\n")
    # print a grid of stars
    cols = 10
    rows = 10

    for r in range(rows):
       # represents one row made up of columns

       for c in range(cols):
           sys.stdout.write("x")
       print()

def triangles():

    # triangle example

    print("\n - - - - - - - - triangles example - - - - - - - - - -\n")
    # print a grid of stars
    cols = 10
    rows = 10
```

```
    spaces = 0

    for r in range(rows):

        #prints the spaces
        for s in range(spaces):
            sys.stdout.write(" ")
        # prints the xs
        for c in range(cols):
            sys.stdout.write("x")
        # end of a row  - drops the cursor
        print()
        cols = cols -1
        spaces = spaces + 1

def os_example():

    # how to call an operating instruction
    # pending your operating system, the commands may differ
    os.system('dir')

def main():
    for_example()
    while_example()

    try_example()

    grid_example()
    triangles()

    os_example()

    print("\n\n\nbye")
main()
```

0.15.3 More Loop Examples

Key Ideas

for loop
for c in range(10):
10 - loops for iterations 0 thru 9
c is the counter, it keeps track of each iteration/loop
for c in range(10,20):
c / counter starts at 10 and ends at 19
for c in range(10,20,2)
c / counter starts at 10 and counts to 19 via 2s
In the examples below, note how the counter is being used to represent different things.

```
# -* - coding: utf -8 -* -
"""
Created on Fri Sep 22 11:44:03 2023
Programmer: James Goudy
Title: Programming Guru

@author: jgoudy

"""
import random
import sys


def loopDaysOfWeek():

    # loop through the days of the week
    # where a number represents a day
    # 0  - Sunday
    # 1  - Monday
    # 2  - Tuesday
    # 3  - Wednesday
    # 4  - Thursday
    # 5  - Frday
    # 6  - Saturday

    # If we do not give a starting number
    # the for statement will the counter (c)
    # with the number 0

    print("\nNUMBER\nLoop through the days of the week using numbers")

    # Enter the number of days.  We are assuming the days start on
    # a Sunday - day 0

    # numOfDays in a week starting at zero
    numOfDays = 6

    for c in range(numOfDays):

        if (c == 0):
            print("c = " + str(c) + "\tSunday")
```

```
            print("\t\tPicnic Day")

        elif (c == 1):
            print("c = " + str(c) + "\tMonday")
            print("\t\tStart of Work Week")

        elif (c == 2):
            print("c = " + str(c) + "\tTuesday")
            print("\t\t2nd day of work week")

        elif (c == 3):
            print("c = " + str(c) + "\tWednesday")
            print("\t\tHump Day ")

        elif (c == 4):
            print("c = " + str(c) + "\tThursday")
            print("\t\tTime to get things done")

        elif (c == 5):
            print("c = " + str(c) + "\tFriday")
            print("\t\tLast day of the work week")

        elif (c == 6):
            print("c = " + str(c) + "\tSaturday")
            print("\t\tIt's the weekend")

        # loop to the top of the list where c is increment to the next number


def dayOfWeek():

    # Find the day of the week
    # where a number represents a day
    #
    # In this example we assume that the
    # days start on Sunday
    # If we entered 8 days, that should give the
    # answer Sunday -  SU M T W R F SA SU - Note that
    # the 8th day starts back on Sunday.
    #
    # 0  - Sunday
    # 1  - Monday
    # 2  - Tuesday
    # 3  - Wednesday
    # 4  - Thursday
    # 5  - Frday
    # 6  - Saturday

    # If we do not give a starting number
    # the for statement will the counter (c)
    # with the number 0

    print("\nNUMBER\nLoop through the days of the week using numbers")
```

```
    # Enter the number of days.  We are assuming the days start on
    # a Sunday - day 0

    # numOfDays in a week starting at zero
    numOfDays = int(input("Enter the numbers of days \n" +
                          " assuming we are starting on Sunday: ")) - 1

    # dividing the number by the 7, the remainder
    # will give us the day of the week.
    # In order to get the remainder, the modulus operator
    # is used %

    c = numOfDays % 7

    if (c == 0):
        print("c = " + str(c+1) + "\tSunday")
        print("\t\tPicnic Day")

    elif (c == 1):
        print("c = " + str(c+1) + "\tMonday")
        print("\t\tStart of Work Week")

    elif (c == 2):
        print("c = " + str(c+1) + "\tTuesday")
        print("\t\t2nd day of work week")

    elif (c == 3):
        print("c = " + str(c+1) + "\tWednesday")
        print("\t\tHump Day ")

    elif (c == 4):
        print("c = " + str(c+1) + "\tThursday")
        print("\t\tTime to get things done")

    elif (c == 5):
        print("c = " + str(c+1) + "\tFriday")
        print("\t\tLast day of the work week")

    elif (c == 6):
        print("c = " + str(c+1) + "\tSaturday")
        print("\t\tIt's the weekend")

        # loop to the top of the list where c is increment to the next number


def loopDaysOfWeek_List():

    print("\nLIST\nLoop through the days of the week using a list")

    weekdays = ["Sunday", "Monday", "Tuesday",
                "Wednesday", "Thursday", "Friday", "Saturday"]
```

```
    # In this case, the for statement only uses the
    # key word "in"
    # c represents a day for each item in the list

    for c in weekdays:

        if (c == "Sunday"):
            print("c = " + str(c) + "\t\tSunday")
            print("\t\t\t\tPicnic Day")

        elif (c == "Monday"):
            print("c = " + str(c) + "\t\tMonday")
            print("\t\t\t\tStart of Work Week")

        elif (c == "Tuesday"):
            print("c = " + str(c) + "\t\tTuesday")
            print("\t\t\t\t2nd day of work week")

        elif (c == "Wednesday"):
            print("c = " + str(c) + "\tWednesday")
            print("\t\t\t\tHump Day ")

        elif (c == "Thursday"):
            print("c = " + str(c) + "\tThursday")
            print("\t\t\t\tTime to get things done")

        elif (c == "Friday"):
            print("c = " + str(c) + "\t\tFriday")
            print("\t\t\t\tLast day of the work week")

        elif (c == "Saturday"):
            print("c = " + str(c) + "\tSaturday")
            print("\t\t\t\tIt's the weekend")

    # loop to the for statement for the next item in the list


def rolladice():

    print("Roll a dice")

    rollagain = "y"

    # while loop is for program control
    # as long as the value of quit is "y"
    # the function will continue to all the user to roll dice

    while (rollagain == "y"):

        # declare some variables
        numberOfRolls = int(input("\nEnter the number of rolls: ")) + 1

        rollValue = 0
```

```
        # c is keeping track of the number of rolls
        # note that 1 is the starting value and
        # numberOfRolls is the ending value

        for c in range(1, numberOfRolls):

            # roll the dice which is created by getting a random number
            # from 1 to 6

            rollValue = random.randint(1, 6)

            print("Roll number " + str(c) +
                  " | dice value is " + str(rollValue))

        rollagain = input("Do you want to roll again y/n? ").lower()

        # loop to the while statement


def countby():

    # below is an example of a
    # multilined string. The """ must
    # be on the same line as the equal sign

    description = """
    COUNT BY

    The purpose of this function
    is to allow the user to enter
    a range of numbers and count
    them by a multiple that they
    select.

    For instance a user enters
    a range starting at 0 and
    ends 10 and they want to count
    by 2\'s the output would like
    the following:

    0 2 4 6 8 10

    """
    print(description)

    runAgain = "y"

    # while loop is for program control
    # as long as the value of quit is "y"
    # the function will continue to all the user to run the function

    while (runAgain == "y"):
```

```
        # ************* VARIABLES FOR THE FOR STATEMENT ***********
        numStart = int(input("Enter your start number: "))
        numEnd = int(input("Enter your end number: "))
        numCountBy = int(input("Enter your countby number: "))

        # ********************************************************

        numCols = int(input("\nEnter the number of " +
                            "columns to display output: "))

        # get the length of the number and add 2 to the length
        numLen = len(str(numEnd)) + 2

        # column count
        colcount = 0

        # print the range using a for statement
        for c in range(numStart, numEnd, numCountBy):

            # print the number
            sys.stdout.write(str(c).rjust(numLen, " "))

            # increment column count by 1
            colcount = colcount + 1

            # track columns and start new row if colcount
            # is greater than numcols value
            if (colcount >= numCols):
                print()
                # reset column count
                colcount = 0

        # End of for loop

        runAgain = input("\n\nDo you want to run again y/n? ").lower()

    # loop to the while statement


def menu():

    mnu = """
    LOOP EXAMPLES
    1. Days of week by number
    2. Calculate day of the week
    3. Days of week by list
    4. Roll a dice
    5. Number Range with Countby (step)
    """

    print(mnu)
```

```
    choice = int(input("Enter 1,2,3 or 4: "))

    if (choice == 1):
        loopDaysOfWeek()
    elif (choice == 2):
        dayOfWeek()
    elif (choice == 3):
        loopDaysOfWeek_List()
    elif (choice == 4):
        rolladice()
    elif (choice == 5):
        countby()
    else:
        print("That wasn't a choice")


def main():
    print("\n")

    quit = "n"

    # while loop for program control
    while (quit != "y"):

        try:

            menu()

            quit = input("Would you like to quit y/n : ")

        except Exception as e:
            print("\n" + str(e))
            print("\nYou had a error, please try again\n")
            quit = input("\nWould you like to quit y/n : ")

    print("\n\nbye")


main()


"""
OUTPUT

    LOOP EXAMPLES
    1. Days of week by number
    2. Days of week by list
    3. Roll a dice
    4. Number Range with Countby (step)

Enter 1,2,3 or 4:

NUMBER
```

```
Loop through the days of the week using numbers
c = 0 Sunday
  Picnic Day
c = 1 Monday
  Start of Work Week
c = 2 Tuesday
  2nd day of work week
c = 3 Wednesday
  Hump Day
c = 4 Thursday
  Time to get things done
c = 5 Friday
  Last day of the work week
Would you like to quit y/n :

LIST
Loop through the days of the week using a list
c = Sunday  Sunday
    Picnic Day
c = Monday  Monday
    Start of Work Week
c = Tuesday  Tuesday
    2nd day of work week
c = Wednesday Wednesday
    Hump Day
c = Thursday Thursday
    Time to get things done
c = Friday  Friday
    Last day of the work week
c = Saturday Saturday
    It's the weekend
Would you like to quit y/n :

Roll a dice

Enter the number of rolls: 4
Roll number 1 | dice value is 4
Roll number 2 | dice value is 3
Roll number 3 | dice value is 5
Roll number 4 | dice value is 1
Do you want to roll again y/n?

    COUNT BY

    The purpose of this function
    is to allow the user to enter
    a range of numbers and count
    them by a multiple that they
    select.

    For instance a user enters
    a range starting at 0 and
    ends 10 and they want to count
```

```
    by 2's the output would like
    the following:

    0 2 4 6 8 10


Enter your start number: 12
Enter your end number: 144
Enter your countby number: 6

Enter the number of columns to display output: 4
   12   18   24   30
   36   42   48   54
   60   66   72   78
   84   90   96  102
  108  114  120  126
  132  138

Do you want to run again y/n?

"""
```

0.15.4 Loops - Typewriter Simulation

This is a fun little exercise.

```
# -* - coding: utf -8 -* -
"""
Created on Thu Sep 15 18:54:58 2022

@author: jgoudy
"""
import sys
import time

winText = "Even though large tracts of Europe " + \
    "and many old and famous States have fallen or " + \
    "may fall into the grip of the Gestapo and " + \
    "all the odious apparatus of Nazi rule, " + \
    "we shall not flag or fail. We shall go on to the end, " + \
    "we shall fight in France, we shall fight on the seas and oceans, " + \
    "we shall fight with growing confidence and " + \
    "growing strength in the air, " + \
    "we shall defend our Island, whatever the cost may be, " + \
    "we shall fight on the beaches, " + \
    "we shall fight on the landing grounds, " + \
    "we shall fight in the fields and in the streets, " + \
    "we shall fight in the hills; we shall never surrender, " + \
    "and even if, which I do not for a moment believe, " + \
    "this Island or a large part of it were subjugated and starving, " + \
    "then our Empire beyond the seas, armed and " + \
    "guarded by the British Fleet, would carry on the struggle, " + \
    "until, in Gods good time, the New World, " + \
    "with all its power and might, steps forth to the rescue and " + \
    "the liberation of the old. - Winston Churchill"


def typeText(someText):

    # program is to approximately 60 chars per line
    # and not split words
    chrcount = 0

    for l in range(len(someText)):

        sys.stdout.write(someText[l])

        chrcount +=1
        if(chrcount > 60 and someText[l] == " "):
            sys.stdout.write("\n")
            chrcount = 0


        time.sleep(.02)
```

```
def main():

    theText = winText;
    typeText(theText)


main()
```

0.15.5 Menu Shell

This is an example of how to create a basic menu system. Note the while statement for program control in the main function.

```
# happy menu

def happyfunction1():
    print("This is happy function 1")

def happyfunction2():
    print("This is happy function 2")

def happyfunction3():
    print("This is happy function 3")


def menu():

    choice = -1

    print("Program choices")
    print("1. Happy Function 1")
    print("2. Happy Function 2")
    print("3. Happy Function 3")
    choice = int(input("Please choose 1, 2, or 3:  "))

    if(choice == 1):
        happyfunction1()
    elif(choice ==2):
        happyfunction2()
    elif(choice == 3):
        happyfunction3()
    else:
        print("That wasn't a choice")


def main():

    choice = "n"

    # program control
    while(choice == "n"):
        menu()

        choice = input("Would you like to quit y/n ")


main()
```

0.16 Itertools

0.16.1 Key Points

- **itertools:** A module providing functions for creating efficient iterators for various tasks.
- **count([start[, step]])**: Creates an infinite iterator that starts at a specified value and increments by a specified step (defaults to 1).
- **cycle(p)**: Creates an infinite iterator that endlessly repeats the elements from an iterable.
- **permutations(iterable, r=None)**: Generates all possible arrangements (permutations) of elements from an iterable, with a specified length if provided. (Order does matter.)
- **combinations(iterable, r)**: Generates all possible combinations of elements from an iterable, with a specified length. (Order doesn't matter unlike permutations)
- **accumulate(p[, func])**: Applies a function (default is addition) cumulatively to elements in an iterable, yielding the sums at each step.
- **filterfalse(predicate, sequence)**: Creates an iterator that filters out elements from an iterable based on a predicate function (keeps elements that return False).
- **lru_cache(maxsize=128, typed=False)**: Creates a function decorator that caches the results of the function based on the arguments (Least Recently Used cache with size limit).
- **batched(p, n)**: Creates an iterator that yields fixed-size chunks of elements from an iterable.

0.16.2 Concept

In computer programming, an iterable is essentially a container that holds elements and allows you to access them one by one in a specific order. Think of it like a box with items you can take out, one at a time.

Here's a breakdown of what iterables are and how they work:

- **Usable with loops:** The key feature of iterables is that you can use them in 'for' loops to process elements efficiently. The loop automatically iterates over the iterable, retrieving each element on each pass.
- **Examples:** Common examples of iterables include lists, tuples, strings, sets, and dictionaries (in some languages).
- **Behind the scenes:** Iterables typically implement a special method (often named `__iter__`) that creates an iterator object. This iterator keeps track of the current position within the iterable and provides access to elements one at a time.

Here's an analogy: Imagine a bookshelf (iterable) containing books (elements). You can't grab all the books at once, but you can go through them one by one (iteration) using a ladder (iterator) to reach each book on the shelf.

Iterables are fundamental for processing sequences of data in programming and form the backbone of many powerful techniques.

0.16.3 Itertools Library

The `itertools` module in Python is a toolbox specifically designed to work with iterables. It provides functions that create new iterators based on existing iterables. These new iterators can perform various manipulations on the original data in a memory-efficient way.

Here is a sample of how they work. This is the accumulate function. Note that it returns an iterator.

The `itertools.accumulate()` function in Python is a powerful tool for performing cumulative operations on iterables. It creates an iterator that yields the sum (by default) or any other function applied cumulatively to the elements of the iterable.

Here's a breakdown of how `itertools.accumulate()` works:

- **Cumulative Operations:** It applies a function to successive elements of the iterable, keeping track of a running total. This total is then yielded by the iterator at each step.

- **Default Behavior (Sum):** If you don't provide a custom function, `accumulate()` performs addition by default. This is useful for calculating running sums, like finding the total sales up to a certain point in a list.
- **Custom Functions:** You can provide a custom function to `accumulate()` to perform different cumulative operations. This function should take two arguments: the current accumulated value and the next element from the iterable. The function's return value becomes the new accumulated value for the next iteration.

Here are some key points to remember about `itertools.accumulate()`:

- ***Returns an Iterator:*** *It's important to note that `accumulate()` returns an iterator, not a list. This means the elements are generated on-demand, saving memory compared to creating a new list to store all the results.*
- **Optional Initial Value (Python 3.8+):** In Python versions 3.8 and later, you can specify an initial value using the `initial` argument. This value will be prepended to the beginning of the accumulated results.

0.16.4 Documentation

https://docs.python.org/3.12/library/itertools.html#

Note there are more itertools listed in the official document

0.16.5 Code

```
"""
Itertools
Programmer: James Goudy
Date: 07/19/2024

This script demonstrates various functionalities from the itertools library in Python.
"""

# Import libraries
import itertools
import operator
import functools

import sys

# Function to generate a count loop (runs indefinitely)
def iter_count():
  """
  This function demonstrates the itertools.count() function
  which generates an unending sequence of numbers.

  Parameters:
      None

  Returns:
      None (prints the sequence to the console)
  """
  print("\nCount Example")

  start = 0
  stop = 500
  step = 100
```

```
  for i in itertools.count(start, step):
    print(i)

    # Break the loop when reaching the stop value
    if i >= stop:
      break

# Function to cycle through elements in a list repeatedly
def iter_cycle():
  """
  This function demonstrates the itertools.cycle() function
  which iterates through a list repeatedly.

  Parameters:
      None

  Returns:
      None (prints the cycled elements to the console)
  """
  print("\nCycle Example")

  cntr = 0
  myList = [10, 20, 30, 40]
  myList2 = ["AA", "BB", "CC", "DD"]
  repeatCycle = 2

  for i in itertools.cycle(myList):
    print(i)
    cntr += 1

    # Break the cycle after a specified number of repetitions
    if cntr >= (repeatCycle * len(myList)):
      break

  cntr = 0
  print()
  for i in itertools.cycle(myList2):
    print(i)
    cntr += 1

    # Break the cycle after a specified number of repetitions
    if cntr >= (repeatCycle * len(myList)):
      break

# Function to generate permutations (ordered arrangements)
def iter_permutations():
  """
  This function demonstrates the itertools.permutations() function
  which generates all possible ordered arrangements (permutations) of elements.

  Parameters:
      None
```

```
  Returns:
      None (prints the permutations to the console)
  """
  print("\nPermutation Example")

  myList = [1, 2, 3]

  for prm in itertools.permutations(myList):
    print(prm)

# Function to generate combinations (unordered arrangements)
def iter_combinations():
  """
  This function demonstrates the itertools.combinations() function
  which generates all possible unordered arrangements (combinations) of elements.

  Parameters:
      None

  Returns:
      None (prints the combinations to the console)
  """
  print("\nCombinations Example")

  myList = [1, 2, 3, 4]

  combLen = len(myList)

  for cSize in range(1, combLen + 1):

    print("\nCombination size = {}".format(cSize))

    for prm in itertools.combinations(myList, cSize):
      print(prm)

# Function to calculate accumulated sums
def iter_accumulate():
  """
  This function demonstrates the itertools.accumulate() function
  which returns a sequence of partial sums.

  Parameters:
      None

  Returns:
      None (prints the accumulated sums to the console)
  """
  print("\nAccumulation Example")

  myList = [2, 4, 5, 10]
  print("Data is - {}".format(myList))
```

```
  # Rolling total using accumulate
  myAcc = itertools.accumulate(myList)
  print(list(myAcc))

  # Iterate through each item in the accumulated sequence
  print("\nIterate through \neach item in Acc")
  for x in itertools.accumulate(myList):
    print("item is {}".format(x))

  print("\nMultiple instead of add")
  myAcc = itertools.accumulate(myList, operator.mul)
  print(list(myAcc))

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Filter out data based on a specific criteria

# Helper function for iter_filterfalse
def divByTwo(a):
  """
  This function checks if a number is odd (not divisible by 2).
  It's a helper function used by iter_filterfalse.

  Parameters:
      a (int): The number to check

  Returns:
      bool: True if the number is odd, False otherwise
  """

def iter_filterfalse():
  """
  This function demonstrates the itertools.filterfalse() function
  which filters out elements based on a condition.

  Parameters:
      None

  Returns:
      None (prints the filtered elements to the console)
  """
  myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

  # Filter out odd numbers (elements not divisible by 2) using divByTwo function
  evenNums = itertools.filterfalse(divByTwo, myList)
  print("Evens", list(evenNums))

  # Filter out even numbers (elements divisible by 2) using lambda function
  evenNums = itertools.filterfalse(lambda x: x % 2, myList)
  print("Evens", list(evenNums))

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# - - - - - - - - - - leave recursion in cache - - - - - - - - - - - - - -
```

```
global countFastSteps
global countSlowSteps

countFastSteps = 0
countSlowSteps = 0

@functools.lru_cache()
def fibfast(n):
  """
  This function calculates the nth Fibonacci number using recursion
  with memoization (lru_cache) for efficiency.

  Parameters:
      n (int): The index of the Fibonacci number to calculate

  Returns:
      int: The nth Fibonacci number
  """
  global countFastSteps
  countFastSteps = countFastSteps + 1
  return fibfast(n -2) + fibfast(n -1) if n > 1 else 1

def fibslow(n):
  """
  This function calculates the nth Fibonacci number using recursion
  without memoization (slower approach).

  Parameters:
      n (int): The index of the Fibonacci number to calculate

  Returns:
      int: The nth Fibonacci number
  """
  global countSlowSteps
  countSlowSteps += 1
  return fibslow(n -2) + fibslow(n -1) if n > 1 else 1

def lru_cache_example(i):
  """
  This function demonstrates the performance improvement of memoization (lru_cache)
  by comparing the number of function calls needed to calculate the nth Fibonacci number
  with and without memoization.

  Parameters:
      i (int): The index of the Fibonacci number to calculate

  Returns:
      None (prints the number of function calls to the console)
  """
  # Initialize variables globally to avoid errors
  global countSlowSteps
  global countFastSteps
```

```
  countSlowSteps = 0
  countFastSteps = 0

  fibfast(i)
  fibslow(i)

  print(countFastSteps)
  print('With lru_cache total function steps: ', countFastSteps)
  print('Without lru_cache total function steps: ', countSlowSteps)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def itr_batch():
  """
  This function demonstrates the itertools.batched() function
  which groups elements from an iterable into fixed -size chunks.

  Parameters:
      None

  Returns:
      None (prints the batched elements and manipulates them to the console)
  """

  print("\nBatch Process")

  # Create a list of numbers
  my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]

  # Use itertools.batched() to group elements in the list into chunks of size 4
  for batch_item in itertools.batched(my_list, 4):
    # Convert the batched item (generator) to a list and print it
    print(list(batch_item))  # Print the batched elements

    # Loop through each item within the current batch
    for item in batch_item:
      # Multiply the item by 10 and print it with a space
      sys.stdout.write(str(item * 10) + " ")
    # Add a newline and separator after processing each batch
    print("\n - - -\n")

  # Define a multi -line string containing a quote
  quote = """
  Never argue with stupid people,
  they will drag you down to their level
  and beat you with experience. - Mark Twain
  """

  # Use itertools.batched() again to group characters in the quote into chunks of size 7
  for chunk in itertools.batched(quote, 7):
    # Print each chunk of the quote
    print(chunk)
def main():
```

```
    iter_count()

    iter_cycle()

    iter_permutations()

    iter_combinations()

    iter_accumulate()

    iter_filterfalse()

    lru_cache_example(25)

    itr_batch()

    print("\n\nbye\n")

main()

,,,
Count Example
0
100
200
300
400
500

Cycle Example
10
20
30
40
10
20
30
40

AA
BB
CC
DD
AA
BB
CC
DD

Permutation Example
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
```

```
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)

Combinations Example

Combination size = 1
(1,)
(2,)
(3,)
(4,)

Combination size = 2
(1, 2)
(1, 3)
(1, 4)
(2, 3)
(2, 4)
(3, 4)

Combination size = 3
(1, 2, 3)
(1, 2, 4)
(1, 3, 4)
(2, 3, 4)

Combination size = 4
(1, 2, 3, 4)

Accumulation Example
Data is - [2, 4, 5, 10]
[2, 6, 11, 21]

Iterate through
each item in Acc
item is 2
item is 6
item is 11
item is 21

Multiple instead of add
[2, 8, 40, 400]
Evens [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Evens [2, 4, 6, 8, 10]
26
With lru_cache total function steps:  26
Without lru_cache total function steps:  242785

Batch Process
[1, 2, 3, 4]
10 20 30 40
 - - -
```

```
[5, 6, 7, 8]
50 60 70 80
 - - -

[9, 10, 11, 12]
90 100 110 120
 - - -

[13, 14, 15, 16]
130 140 150 160
 - - -

[17]
170
 - - -

('\n', ' ', ' ', 'N', 'e', 'v', 'e')
('r', ' ', 'a', 'r', 'g', 'u', 'e')
(' ', 'w', 'i', 't', 'h', ' ', 's')
('t', 'u', 'p', 'i', 'd', ' ', 'p')
('e', 'o', 'p', 'l', 'e', ',', ' ')
('\n', ' ', ' ', 't', 'h', 'e', 'y')
(' ', 'w', 'i', 'l', 'l', ' ', 'd')
('r', 'a', 'g', ' ', 'y', 'o', 'u')
(' ', 'd', 'o', 'w', 'n', ' ', 't')
('o', ' ', 't', 'h', 'e', 'i', 'r')
(' ', 'l', 'e', 'v', 'e', 'l', ' ')
('\n', ' ', ' ', 'a', 'n', 'd', ' ')
('b', 'e', 'a', 't', ' ', 'y', 'o')
('u', ' ', 'w', 'i', 't', 'h', ' ')
('e', 'x', 'p', 'e', 'r', 'i', 'e')
('n', 'c', 'e', '.', ' ', ' -', ' ')
('M', 'a', 'r', 'k', ' ', 'T', 'w')
('a', 'i', 'n', '\n', ' ', ' ')


bye
'''
```

0.17 Recursion

0.17.1 Explanation

Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself. It is a powerful technique that can be used to solve a wide variety of problems, from mathematical to computational.

A recursive function is a function that calls itself directly or indirectly. This means that the function can solve a problem by breaking it down into smaller subproblems, each of which is solved using the same function.

Recursion works by having two cases:

- **Base case:** This is the simplest version of the problem, which can be solved directly without using recursion.
- **Recursive case:** This is the case where the problem is broken down into smaller subproblems, each of which is solved using the recursive function itself.

The recursive function will continue calling itself until it reaches the base case, at which point it will return the solution to the problem.

Example:

```
def factorial(n):
    if n == 0:
        # This is the base case n == 0
        return 1
    else:
        return n * factorial(n - 1)

def main():

    num = int(input("Enter a whole number: "))

    ans = factorial(num)

    # note that we can use {} with the .format as placeholders
    print("The factorial of {} is {}".format(num,ans))

main()
```

This function calculates the factorial of a number, which is the product of all the positive integers less than or equal to that number.

The base case is when n is 0, in which case the factorial is 1. Otherwise, the recursive case breaks down the problem into calculating the factorial of n - 1, and then multiplying that result by n.

For example, to calculate the factorial of 5, the function would first call itself to calculate the factorial of 4. This would return 24. The function would then multiply 24 by 5, to return the final answer of 120.

Recursion can be a difficult concept to grasp at first, but it is a very powerful technique that can be used to solve a wide variety of problems.

Here are some other examples of problems that can be solved using recursion:

- Finding the maximum or minimum value in a list
- Searching for an element in a list or tree
- Traversing a tree or graph
- Solving the Towers of Hanoi puzzle
- Implementing a quicksort algorithm

0.17.2 Drawbacks To Recursion

Recursion is a powerful programming technique that can be used to solve many complex problems in a concise and elegant way. However, it also has some drawbacks:

- **Memory usage:** Recursive functions typically use more memory than iterative functions, because each recursive call creates a new stack frame. This can be a problem for large or complex problems, or for programs that need to run on devices with limited memory.
- **Stack overflow:** If a recursive function is not implemented correctly, it can lead to a stack overflow error. This happens when the call stack exceeds its allocated size, which can cause the program to crash.
- **Difficulty in understanding and debugging:** Recursive code can be difficult to understand and debug, especially for inexperienced programmers. This is because it can be challenging to track the flow of execution and to identify the different cases that the function is handling.
- **Slower execution time:** Recursive functions are often slower than iterative functions, because of the overhead involved in calling the function itself. This can be a problem for performance-critical applications.
- **Limited applicability:** Recursion is not suitable for all problems. Some problems have more efficient iterative solutions, and others may not lend themselves well to recursive solutions at all.

Overall, recursion is a powerful tool that can be used to solve many complex problems. However, it is important to be aware of the drawbacks before using it.

Here are some tips for using recursion effectively:

- **Only use recursion when it is the most efficient or elegant solution.** There are many problems that can be solved both recursively and iteratively (loops). In general, iterative(loops) solutions are more efficient and easier to understand.
- **Be careful of stack overflow.** Make sure that your recursive functions have a base case that will eventually be reached, and that they do not make recursive calls too many times.
- **Write clear and concise recursive code.** Use comments to explain what your code is doing and why.
- **Test your recursive code thoroughly.** Make sure that it works correctly for all possible inputs.

Source: Google Bard / Edited and Checked by James Goudy

0.17.3 Example 1 Bottles Of Beer

```
def BottlesOfBeer(num):

    if num > 0:
        temp = num -1
        print(str(num) + "  bottles of rootbeer on the wall "
              + str(num) + " bottles of rootbeer")
        print("take one down pass it around "
              + str(temp) + " bottles of rootbeer on the wall")

        BottlesOfBeer(num - 1)

def main():
    numbottles = 50
    BottlesOfBeer(numbottles)

main()
```

0.17.4 Example 2 Sum Of List

```
def sumlist(numberList):
```

```
    if len(numberList) == 1:
        return numberList[0]
    else:
        return numberList[0] + \
               sumlist(numberList[1:])

def main():

    theList = [1,3,5,7,9]

    ans = sumlist(theList)

    print("The sum of {} is {}".format(theList,ans))

main()
```

0.17.5 Example 3 Sierpiski triangle

Note: In Spyder set graphics to Automatic

```
import turtle

def drawTriangle(points,color,myTurtle):
    myTurtle.fillcolor(color)
    myTurtle.up()
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.down()
    myTurtle.begin_fill()
    myTurtle.goto(points[1][0],points[1][1])
    myTurtle.goto(points[2][0],points[2][1])
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.end_fill()

def getMid(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

def sierpinski(points,degree,myTurtle):
    colormap = ['blue','red','green','white','yellow',
                'violet','orange']
    drawTriangle(points,colormap[degree],myTurtle)
    if degree > 0:
        sierpinski([points[0],
                        getMid(points[0], points[1]),
                        getMid(points[0], points[2])],
                   degree -1, myTurtle)
        sierpinski([points[1],
                        getMid(points[0], points[1]),
                        getMid(points[1], points[2])],
                   degree -1, myTurtle)
        sierpinski([points[2],
                        getMid(points[2], points[1]),
                        getMid(points[0], points[2])],
                   degree -1, myTurtle)
```

```
def main():
   myTurtle = turtle.Turtle()
   myWin = turtle.Screen()
   myPoints = [[ -100, -50],[0,100],[100, -50]]
   myPoints = [[ -150, -100],[0,150],[150, -100]]
   sierpinski(myPoints,3,myTurtle)


   turtle.exitonclick()
   turtle.done()
   turtle.bye()

main()
```

0.17.6 Example 4 Tree

```
import turtle

def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(25)
        tree(branchLen -10,t)
        t.left(50)
        tree(branchLen -10,t)
        t.right(25)
        t.backward(branchLen)

def main():
    t = turtle.Turtle()
    t.speed(0)
    myWin = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75,t)
    myWin.exitonclick()
    myWin = None
    t = None


    turtle.exitonclick()
    turtle.done()
    turtle.bye()

main()
```

0.17.7 Example 5 - Recursion / Loop / Voice - BOB

```
# -* - coding: utf -8 -* -
"""
```

```
Created on Fri Feb 25 15:06:44 2022

@author: jgoudy

James Goudy

Text To Speach
https://pypi.org/project/pyttsx3/

Do not use PIP witih ANACONDA
choose pyttsx3 via the Anaconda Navigator

NOTE: You can run this in IDLE if you
install pip and install
    pip install pyttsx3

"""

import pyttsx3

# global variables
se = pyttsx3.init()
n = 1

# set the number of bottles


def numBottles():

    global n

    try:

        n = int(input("Enter the number of bottles"))

    except:
        n = 3

# text to speech


def sayBottles(numBottles):

    cntr = numBottles

    while cntr > 0:

        se.say(str(cntr) + " bottles of beer on the wall " + str(cntr) +
               " bottles of beer. Take one down pass it around " +
               str(cntr -1) + " bottles of beer on the wall")
        se.runAndWait()
        cntr -= 1
```

```
# print the number of bottles using a loop
def bottleloop(numBottles):

    cntr = numBottles

    while cntr > 0:

        print(str(cntr) + " bottles of beer on the wall " + str(cntr) +
              " bottles of beer. Take one down pass it around " +
              str(cntr -1) + " bottles of beer on the wall")
        cntr -= 1


# print the number of bottles usinga recursive loop
def bottlerecursive(numBottles):

    # base case - base case is the condition that stops the loop
    if numBottles < 1:
        return

    print(str(numBottles) + " bottles of beer on the wall " +
          str(numBottles) +
          " bottles of beer. Take one down pass it around " +
          str(numBottles -1) + " bottles of beer on the wall")

    bottlerecursive(numBottles -1)

# play the voices and changle thevoice


def playVoices():

    vchoice = 0

    # get the voices and store them in a list
    v = se.getProperty('voices')

    # print the voices info
    print(v)

    # setProperty is used to change the voice
    for i in range(len(v)):
        se.setProperty("voice", v[i].id)

        se.say("This is voice " + str(i))
        se.runAndWait()

    vchoice = int(input("Please choose a voice"))

    se.setProperty("voice", v[vchoice].id)

# menu system
```

```
def menu():

    choice = 0

    menuString = "\n\n1. loop version of Bottles of Beer\n" + \
        "2. Recursive version of Bottles of Beer\n" + \
        "3. Spoken version of Bottles of Beer\n" + \
        "4. Change voice\n" + \
        "5. Cancel\n"

    print(menuString)

    choice = int(input("Please choose 1,2,3,4,5 : "))

    if (choice == 1):

        numBottles()
        bottleloop(n)

    elif (choice == 2):

        numBottles()
        bottlerecursive(n)

    elif (choice == 3):

        numBottles()
        sayBottles(n)

    elif (choice == 4):

        playVoices()

    else:

        return


def main():

    quit = "n"

    while quit != "y":

        # note that a try statement will not tell you
        # where you have a problem.  To be granular, you would
        # need place the  try statment within the individual functions
        try:

            menu()
```

```
            quit = input("Would you like to quit? y/n ").lower()

        except:

            print("There was an error try again")

    # notify the user the program is doone
    print("\nbye bye")


main()
```

0.17.8 Recursion - Loops vs Recursion

Recursion vs. Loops: A Comparative Analysis

Recursion and loops are both programming techniques used to execute code multiple times. However, they differ in their approach and implementation.

Recursion

- **Definition:** A recursive function is a function that calls itself directly or indirectly.
- **Mechanism:** It breaks down a problem into smaller, similar subproblems and solves each subproblem recursively.
- **Termination:** A base case is defined to stop the recursion and prevent an infinite loop.
- **Example:** Factorial calculation, Fibonacci sequence.

Loops

- **Definition:** A loop is a programming construct that repeatedly executes a block of code until a certain condition is met.
- **Types:** While loops, for loops, do-while loops.
- **Mechanism:** The loop condition is evaluated at the beginning or end of each iteration.
- **Example:** Iterating over an array, counting numbers.

Comparison Table:

Feature	Recursion	Loops
Mechanism	Function calls itself	Iterates over a block of code
Termination	Base case	Loop condition
Performance	Can be slower due to function calls	Generally faster
Readability	Can be harder to understand for complex problems	Often more straightforward
Use Cases	Divide-and-conquer algorithms, tree and graph traversal	Iterating over collections, performing repetitive tasks

When to Use Which:

- **Recursion:**
  - Problems that can be naturally divided into smaller, similar subproblems.
  - Tree and graph traversal.
  - Divide-and-conquer algorithms.
- **Loops:**
  - Iterating over collections (arrays, lists, etc.).
  - Performing repetitive tasks with a known number of iterations.
  - Simple, straightforward calculations.

Note: In some cases, both recursion and loops can be used to solve the same problem. However, one approach might be more efficient or easier to understand than the other depending on the specific circumstances. It's often a good practice to consider both options and choose the one that best suits the problem at hand.

```
import datetime
# Developer: James Goudy
# Date: 9/13/24

# The purpose of this program is to show that
# recursion uses memory resources. It also
```

```
# compares this to using a traditional loop.

# global variables
gcntr = 0
lcntr = 0

def runLoop():

        # variables
        loopCntr = 1
        run = True
        choice = "y"

        while(run):

            global lcntr

            # increment counter
            lcntr += 1

            # print count
            print(f"c Loop counter: {lcntr}")

            # check to see if user wants to stop
            # stops every 100,000
            if(lcntr % 100000 == 0):
                choice = input("Continue y/n : ").lower()
                if(choice == "n"):
                    run = False


def runLoopsByTime(secsToRun):

    run = True
    global lcntr

    startTime = datetime.datetime.now()
    endTime = datetime.datetime.now()

    try:
        while(run):

            # increment counter
            lcntr += 1

            # print counter
            print(f"t Loop Counter: {lcntr}")

            endTime = datetime.datetime.now()

            mytimeDif = int((endTime - startTime).total_seconds())

            if(mytimeDif >= secsToRun):
```

```
                run = False

    except Exception as e:
            print(f"\n*** error***\n{e}\n")
            print(f"Max number of loops: {gcntr}")
            input("Press enter to continue")

def runRecursion():

    global gcntr
    run = True

    # this needs to be in try statement.
    # eventually, the resources will run out
    # and cause a crash. the try prevents this.
    # the purpose here is to show the limited
    # resources and purposely run them out
    # so you are aware of this condition.

    try:

        if(run):
            # increment counter
            gcntr = gcntr + 1

            # print count
            print(f"r Recursion counter: {gcntr}")

            # call itself
            runRecursion()
    except Exception as e:
            print(f"\n*** error***\n{e}\n")
            print(f"Max number of recursive loops: {gcntr}")
            input("Press enter to continue:  ")

def summary():

    global gcntr
    global lcntr

    print(f"Recursion ran" + str(gcntr).rjust(9," ") + " times")
    print(f"Loops     ran" + str(lcntr).rjust(9," ") + " times")


def menu():

    choice = -1


    menuText = """
    Recursion Example
    1. Count loops by increments of 100,000
    2. Count loops by time in seconds
```

```
    """
    print(menuText)

    choice = int(input("Enter 1 or 2: "))

    if(choice == 1):
        runRecursion()
        runLoop()
    else:
        secs = int(input("Enter number of secs to run: "))
        runRecursion()
        runLoopsByTime(secs)




def main():

    mySecs = 120

    menu()

    summary()

    print("\nbye\n")

main()
```

Analysis of the Code:

This code provides a basic example of loops and recursion in Python. Here's a breakdown of each function:

1. **Global Variables:**

- `gcntr`: This global variable keeps track of the number of times the `runRecursion` function is called.
- `lcntr`: This global variable tracks the number of iterations in both `runLoop` and `runLoopsByTime` functions.

2. **Functions:**

- `runLoop()`:
  - This function uses a `while` loop to continuously increment the `lcntr` and print its value.
  - It checks every 100,000 iterations if the user wants to continue (stops on "n").
- `runLoopsByTime(secsToRun)`:
  - It uses a `while` loop to keep iterating as long as `run` is True.
  - It increments `lcntr` and prints its value.
  - It calculates the elapsed time using `datetime` and stops the loop if it reaches `secsToRun`.
- `runRecursion()`:
  - This function demonstrates recursion by calling itself within the loop.

```
        **Important Note:** This version deliberately lacks a base case, causing an intentio
```

```
- It increments ‘gcntr‘ and prints its value.
```

- ‘summary()‘:

  - This function simply displays the final count of both ‘gcntr‘ and ‘lcntr‘.

- ‘menu()‘:

  - This function displays a menu and takes user input (1 or 2).
  - Based on the choice, it calls ‘runRecursion‘ and then either ‘runLoop‘ or ‘runLoopsByTime‘

- ‘main()‘:

  - This function sets a default time (120 seconds) for ‘runLoopsByTime‘.
  - It calls the ‘menu‘ function to get user input.
  - Finally, it calls ‘summary‘ to display the final counts and prints a goodbye message.

Key Points:

- The code uses global variables, which can be a bad practice for larger projects as it can l
- The ‘runRecursion‘ function lacks a base case, leading to an infinite loop and eventually a
- The code demonstrates how loops and recursion can be used for counting iterations.

Overall:

This code provides a basic introduction to loops and recursion. However, it’s important to und

0.17.9 Visualizing Recursive Function Calls:

A Stack-Based Approach

Understanding Recursion: A recursive function is a function that calls itself directly or indirectly. This creates a stack of function calls, each with its own set of variables and parameters.

The Stack Data Structure: A stack is a data structure that follows the Last-In-First-Out (LIFO) principle. This means the last element added to the stack is the first one to be removed.

Visualizing the Stack: To visualize how resources are managed during a recursive function call, we can use a stack diagram. Each function call is represented as a stack frame. A stack frame contains:

- **Function name:** The name of the function being called.
- **Parameters:** The values passed to the function.
- **Local variables:** Variables declared within the function.
- **Return address:** The memory address to return to when the function finishes.

Example: Factorial Function
Consider the following recursive factorial function:
Python

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

Stack Diagram for factorial(3):

1. **Initial call:**
   - `factorial(3)` is called.
   - A new stack frame is created with `n = 3`.
2. **Recursive call:**
   - `factorial(2)` is called within `factorial(3)`.
   - A new stack frame is created with `n = 2`.
3. **Recursive call:**
   - `factorial(1)` is called within `factorial(2)`.
   - A new stack frame is created with `n = 1`.
4. **Base case:**
   - `factorial(0)` is called within `factorial(1)`.
   - The base case is reached, and the function returns 1.
5. **Return values:**
   - `factorial(1)` returns 1 to `factorial(2)`.
   - `factorial(2)` calculates 2 * 1 = 2 and returns 2 to `factorial(3)`.
   - `factorial(3)` calculates 3 * 2 = 6 and returns 6 to the main program.

Visual Representation:

```
| - - - - - - - - - - - - - - - -|
| factorial(3)   |
| n = 3         |
| - - - - - - - - - - - - - - - -|
| factorial(2)   |
```

```
| n = 2         |
| - - - - - - - - - - - - - - - -|
| factorial(1)   |
| n = 1         |
| - - - - - - - - - - - - - - - -|
| factorial(0)   |
| n = 0         |
| - - - - - - - - - - - - - - - -|
```

Key Points:

- Each recursive call adds a new stack frame to the top of the stack.
- When a function finishes, its stack frame is removed from the stack.
- The return address in each stack frame ensures that the program can return to the correct location after a function call.
- The stack helps manage the memory and control flow during recursive function calls.

By understanding this stack-based approach, you can better visualize and analyze recursive algorithms.

0.18 Lists

0.18.1 Definition

List: A collection of data values where each value is identified by an index. It is mutable (values can be modified). The values that make up the list are called elements. A data value can be a value, a list, a dictionary, and object, etc.

Designated by the square brackets [ ]

0.18.2 Code

Example 1 - The fundamentals

```
import sys
import random

"""
LISTS - Lecture 1
"""


def ListExample():
    # Create Lists
    print("\nCreating Lists\n")

    aaa = 1
    bbb = 2
    ccc = 3
    ddd = 4
    eee = 5

    # or written as a list
    mynums1  = [1,2,3,4,5]
    print(mynums1)

    stooges = ["Larry", "Curly", "Moe"]
    print(stooges)

    # also we can use variables
    mynums2 = [aaa, bbb, ccc, ddd, eee]
    sys.stdout.write("mynums2 = " + str(mynums2) + "\n")

    # NOT RECOMMENDED AND NOT ALLOWED IN MOST OTHER LANGUAGES
    mixlist = ["car", 42, .5555, True, stooges]
    print(mixlist)

    # we can create an empty list
    # used when we want to have a place to store items later
    emptylist  = []

    # Characteristics
    # Length
```

```
print(len(mynums1))

# Type
print(type(mynums1))

# Accessing Elements - indexes start at 1
print("\nAcessing Elements\n")
print(stooges[1])           # Curly prints
print(stooges[ -1])          # Count from the back - Moe
print(mynums1[0])           # first position - prints 1
print(mynums1[ -2])          # counts to from the rear - prints 4



# check membership
print("\nCheck Membership\n")
print("Shemp" in stooges)
print("Moe" in stooges)
print(42 in mixlist)

# concantenation
print("\nConcantenation")
stooges = stooges + ["Shemp", "Curly Joe"]
mynums1 = mynums1 + [6,7,8,9,10]
print(stooges)
print (mynums1)

# create multiples
mynums3 = [42] * 5      # create 5 42's
print(mynums3)

# slicing - first number is inclusive, the last one is not
print("\nSlicing\n")
print(mynums1[1:4])                    # print numbers 2,3,4
print(mynums1[:3])                     # print first 3 numbers
print(mynums1[: -3])                    # remove last 3 numbers
print(mynums1[:])                      # print all numbers
print(mynums1[0:7:2])                  # print 1,3,5,7
print(mynums1[0:len(mynums1):3])       # print 1,4,7,10
print(mynums1[len(mynums1):0: -2])      # print 10,8,6,4,2

# mutable - change data
print("\nMutable - change data")
stooges[0] = "Tom"
print(stooges)              # ['Tom', 'Curly', 'Moe', 'Shemp', 'Curly Joe']
mynums3[3] = 99
print(mynums3)              # [42, 42, 42, 99, 42]

# deleting items
print("\ndelete items\n")
del stooges[0]              # delete item - tom
print(stooges)
del mynums3[3:5]            # delete last 2 -  last number is not inclusive
```

```
print (mynums3)
print(mynums2)
del mynums2                 # del the whole list
# print(mynums2)            # uncomment this to show the error

# object references
print("\nobject references\n")
fruita = ["apples", "bananas", "mangoes", "tomatoes"]
print(fruita)
fruitb = fruita             # fruitb is looking at the data of fruita
print(fruitb)

fruitb[1] = "oranges"       # change it in b which is really a
print(fruita)               # see the data change in a

# check to see if fruita is actuall fruitb
print(fruita is fruitb)
print(fruita == fruitb)



#clone a copy
print("\nclone data\n")
fruitb = fruita[:]          # using slicing - [:] goes through all items
fruita[1] = "grapes"
print(fruita)
print(fruitb)
print(fruita is fruitb)
print(fruita == fruitb)

# list methods
# Appends adds item to end of list
print("\nlist methods\n")
mylist = []
mylist.append(30)
mylist.append(50)
mylist.append(40)
mylist.append(60)
sys.stdout.write("Appended \t\t" + str(mylist) + "\n")

# sort list
mylist.sort()
sys.stdout.write("Sorted \t\t\t" + str(mylist) + "\n")

# reverse the order of the list
mylist.reverse()
sys.stdout.write("Reversed \t\t" + str(mylist)  + "\n")
mylist.reverse()
sys.stdout.write("Reversed \t\t" + str(mylist)  + "\n")

# insert - parameters (position, value)
mylist.insert(0,10)         #insert at the beginning
sys.stdout.write("Insert 10 \t\t" + str(mylist)  + "\n")
```

```
mylist.insert(2,35)         #insert at the third position
sys.stdout.write("Insert 35 \t\t" + str(mylist)  + "\n")

# retrieve - index retrieves position of specified value
print("35 is index \t\t" + str(mylist.index(35)))
print("50 is index \t\t" + str(mylist.index(50)))

# count - counts the number of a specified item
thevalue = 5
mylist.insert(0,thevalue)
mylist.insert(2,thevalue)
mylist.insert(4,thevalue)
print("Insert 3 " + str(thevalue) + "\'s \t\t" + str(mylist))

cnt = mylist.count(thevalue)       #count the number of fives
# example shows good use of variables and escape characters
print("There are " + str(cnt)
        + " \"" + str(thevalue) + "\'s\" in the list" )

# remove last item
print("The list \t\t" + str(mylist))
mylist.pop()
print("remove last item \t" + str(mylist))

myvar = mylist.pop()           #remove last item and store value in variable
print("myvar = " + str(myvar))
print("The list \t\t" + str(mylist))


# remove the first oocurence of an item
mylist.remove(thevalue)
print("The first "+ str(thevalue) + " removed \t" + str(mylist))

# remove the rest of the cntvalue in the list
for i in mylist:
    if thevalue == i:
        mylist.remove(thevalue)

print("The "+ str(thevalue) + " are removed \t" + str(mylist))

# Extend = add items from a list or iterable object to a list
mylist2 = [111,222,333]
#NOTE if mylist2 is appended it puts mylist2 as an item the list
mylist.append(mylist2)
print("\nappend list to mylist \t" + str(mylist))
mylist.pop()
mylist.extend(mylist2)
print("extend list to mylist \t" + str(mylist))


#list traveral
print("\nList Traversal")
mydata = []
```

```
for cntr in range(0,12):
    mydata.append(random.randint(0,101))
print(mydata)


#iterate by object / item
print("\niterate by item")
for i in mydata:
    sys.stdout.write(str(i) + " ")
print()

#iterate by position
print("\niterate by position")
for position in range(len(mydata)):
    sys.stdout.write(str(mydata[position]) + " ")
print()

#iterate position print every third
print("\nprint every 3rd")
cntr = 1
for position in range(len(mydata)):
    if (cntr % 3) == 0:
        sys.stdout.write(str(mydata[position]) + " ")
        cntr = 0
    cntr += 1
print()

# spit
print("\nSplitting Strings")

# note = make sure you use straight quotes
quote = "Python is always fun"
words = quote.split(" ")

print(words)
print()

# show how this will include spaces and the new line character
quote = """Always code as if the guy who ends up maintaining your
            code will be a violent psychopath who knows where you live"""

words = quote.split(" ")
print(words)
print()

newlist = []
a=""
# clean up long quotes - remove empty spaces and newlines
for i in range(0, len(words) - 1):
    if words[i] == '':
        a = ''                          # python expects a code of line
    elif words[i] == '\n':
```

```
            a= ''
        else:
            newlist.append(words[i])

    print(newlist)

    #clear - remove the data of a list
    newlist.clear()
    print(newlist)

def main():
    ListExample()


main()
```

Example 2 - Passing and Returning Lists via Functions

```
import sys

# -* - coding: utf -8 -* -
"""
    Functions that product lists

    initialize a result variable to be an empty list
    loop
       create a new element
       append it to result
    return the result



"""

def numbers_within_a_range(start,finish):
    # pass a starting and ending number in to the function
    result = []

    # create a list of numbers using the start and finish variables
    for i in range(start,finish):
        result.append(i)

    # return the list to the calling function
    return result

def print_list(data):

    # passs a list into a function and iterate through it
    for i in range(len(data)):
        sys.stdout.write(str(data[i]) + " ")

    print()

def main():
```

```
    mynums = []

    beginning = int(input("Enter start number  "))

    #remember last number in range is not inclusive
    end = int(input("Enter end number  ")) + 1



    mynums = numbers_within_a_range(beginning, end)
    print_list(mynums)


    # Add 10 to every number
    # note we adding this to a new list
    mynums2 = [item+10 for item in mynums]


    print_list(mynums2)


main()
```

Pick Data From A List

```
# -* - coding: utf -8 -* -
"""
@author: jgoudy
"""

List1 = ["Billings", "Boulder", "Bozeman", "Helena", "Kalispell", "Whitefish"]

def pickOne():

    global List1

    for cntr in range(len(List1)):
        # Remember positions start at zero
        # we add one to the cntr because most lists start with 1
        # note that we can use the cntr to create an numbered list
        print(str(cntr+1) + ". " + List1[cntr])

    # Remember to subtract one, since we added one to display
    choice = int(input("Pick the number of the city: ")) - 1

    # retreive the item by position and store it in variable item
    item = List[choice]

    print("You chose " + item)
```

```
def main():

    choice  = "n"

    # writing the != means the only waay to exit the loop is to
    # enter the letter "y"
    while( choice != "y"):
        pickOne()
        choice = input("Would you like to quit? y/n  ").lower()

    print("bye bye")


main()
```

Generate Random Numbers

```
# -* - coding: utf -8 -* -
"""
Created on Mon Oct 14 00:25:09 2019

@author: jgoudy
"""
import random

def pickARandomNumber():

    rn = 0


    for cntr in range(20):
        rn = random.randint(100, 200)

        print("The random number was " + str(rn))

    print(" - - - - - - - - - - - - - - - - - - - - -")

    for cntr in range(20):
        rn = random.randint(10, 20)

        print("The random number was " + str(rn))


def main():
    pickARandomNumber()

main()
```

Split Columns Into Lists

```
# -* - coding: utf -8 -* -
"""
Created on Sun Oct 13 15:09:46 2019
```

```
@author: jgoudy
"""

# Global Variables
text = """
  artless             base -court              apple -john
        bawdy                bat -fowling             baggage
        beslubbering         beef -witted             barnacle
        bootless             beetle -headed           bladder
        churlish             boil -brained            boar -pig
        cockered             clapper -clawed          bugbear
        clouted              clay -brained            bum -bailey
        craven               common -kissing          canker -blossom
        currish              crook -pated             clack -dish
        dankish              dismal -dreaming         clotpole
        dissembling          dizzy -eyed              coxcomb
        droning              doghearted              codpiece
        errant               dread -bolted            death -token
        fawning              earth -vexing            dewberry
        fobbing              elf -skinned             flap -dragon
        froward              fat -kidneyed            flax -wench
        frothy               fen -sucked              flirt -gill
        gleeking             flap -mouthed            foot -licker
        goatish              fly -bitten              fustilarian
        gorbellied           folly -fallen            giglet
        impertinent          fool -born               gudgeon
        infectious           full -gorged             haggard
        jarring              guts -griping            harpy
        loggerheaded         half -faced              hedge -pig
        lumpish              hasty -witted            horn -beast
        mammering            hedge -born              hugger -mugger
        mangled              hell -hated              joithead
        mewling              idle -headed             lewdster
        paunchy              ill -breeding            lout
        pribbling            ill -nurtured            maggot -pie
        puking               knotty -pated            malt -worm
        puny                 milk -livered            mammet
        qualling             motley -minded           measle
        rank                 onion -eyed              minnow
        reeky                plume -plucked           miscreant
        roguish              pottle -deep             moldwarp
        ruttish              pox -marked              mumble -news
        saucy                reeling -ripe            nut -hook
        spleeny              rough -hewn              pigeon -egg
        spongy               rude -growing            pignut
        surly                rump -fed                puttock
        tottering            shard -borne             pumpion
        unmuzzled            sheep -biting            ratsbane
        vain                 spur -galled             scut
        venomed              swag -bellied            skainsmate
        villainous           tardy -gaited            strumpet
        warped               tickle -brained          varlot
        wayward              toad -spotted            vassal
```

```
        weedy               unchin -snouted          whey -face
        yeasty              weather -bitten          wagtail
"""


newList = []

firstColumn = []
secondColumn = []
thirdColumn = []

# - - - - - - - - - - - - End of Global Variables - - - - - - - - - - - - - - - - - - - - - - - -

# split the text on spaces
def splitList(text):

    textList = text.split(" ")

    # use the global variable
    global newList

    # loop through each chunk in the texlist and remove empty spaces
    # and the new lines

    for chunk in textList:

        if(chunk != ''):
            newWord = ""

            # loop through the chunk and build the word
            # character by character but not including the
            # "\n" the new word
            for pos in range(len(chunk)):
                check = chunk[pos]
                if(check != "\n"):
                    newWord = newWord + check
            #append the new word to the new list
            #newlist now contains clean data
            newList.append(newWord)

# divid the list into the three original lists
def divideIntoThree():

    global newList
    global firstColumn, secondColumn, thirdColumn

    cntr = 1;


    #Every third is the start of a new column
    for pos in range(len(newList)):

        # if we divid by three the remainders will be 1,2 and 0
```

```
        # using the modulus "%" we can get the 1, 2 and 0

        if((cntr % 3) == 1):
            firstColumn.append(newList[pos])
        elif((cntr % 3) == 2):
            secondColumn.append( newList[pos])
        else:
            thirdColumn.append(newList[pos])

        cntr += 1

        # reset the counter back to one
        if(cntr == 4):
            cntr = 1



def main():
    splitList(text)
    divideIntoThree()



    print (newList)

    print()

    print(firstColumn)

    print()

    print(secondColumn)

    print()

    print(thirdColumn)



main()
```

0.19 Classes

0.19.1 Key Ideas

- classes
- objects
- inheritance

Definition
Class is code that acts like a blueprint. "Objects" are made from classes. Classes can represent things in terms of code. For example, they could represent a *car* - make, model, year; a *dog* - dog name, breed, color; a *person* - first name, last name, address, city, state; etc.

0.19.2 Lecture Code

Example 1 - Cars

```
# -* - coding: utf -8 -* -
"""
Created on Mon Oct 2019

@author: jgoudy
"""

class car:



    carspeed = 0
    carengineOn = True


    # constructor  - there can only be one in python
    def __init__(self, carmake="",carmodel="",caryear="", carcolor=""):
        self.carspeed = 0
        self.carengineOn = False
        self.carmake = carmake
        self.carmodel = carmodel
        self.caryear= caryear
        self.carcolor = carcolor

    # __str__  defines what print() or str() is printed for the obje
    # example print(car) = what is defined here is what will be printed
    def __str__(self):
        return ("Make: " + self.carmake + "\n" + \
                "Model: " + self.carmodel + "\n" + \
                "Year: " + self.caryear + "\n" + \
                "Color: " + self.carcolor + "\n")


    # __str__ must return string object
    # whereas __repr__ can return any python expression.
    #https://docs.python.org/3/reference/datamodel.html
    # when possible it should return a string
```

```
def __repr__(self):
    return (self.carmake+","+ self.carmodel+","+ self.caryear+","+ self.carcolor)


@property
def make(self):
    return self.carmake
@make.setter
def make(self, value):
    self.carmake = value

@property
def model(self):
    return self.carmodel
@model.setter
def model(self, value):
    self.carmodel = value

@property
def year(self):
    return self.caryear
@year.setter
def year(self, value):
    self.caryear = value

@property
def color(self):
    return self.carcolor
@color.setter
def color(self, value):
    self.carcolor = value

@property
def speed(self):
    return self.carspeed

@property
def engineOn(self):
    return self.carengineOn

def startCar(self):
    self.carengineOn = True

def stopCar(self):
    self.carengineOn = False


def accelerate(self,speed = 5):
    if (self.carengineOn == True) :
        self.carspeed += speed
        print("aaargh")
    else:
```

```
            print("Start the car")


    def deccelerate(self,speed = 5):
        if self.carspeed > 0:
            self.carspeed -= speed
        else:
            self.carspeed = 0
        print("car is stopped")


    def honk(self):
        print("beep beep")
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def main():

   mycar1 = car()

   mycar2 = car("Ford", "Mustang","1967","red")

   mycar2.accelerate()
   mycar2.startCar()
   mycar2.accelerate()
   print(mycar2.speed)
   mycar2.accelerate()
   print(mycar2.speed)
   mycar2.accelerate(50)
   print(mycar2.speed)
   mycar2.deccelerate(70)

   print("The color of my car is " + mycar2.color)
   mycar2.color = "blue"
   print("The color of my car is " + mycar2.color)

   mycar1.honk()

   print("\bye bye")

main()
```

Example 2 - Person / Worker / Voter

```
class Person:
    """ Represents a person. """

    # python can only have one constructor
    def __init__(self, first_name="", last_name="", town=""):
        self.first_name = first_name
        self.last_name = last_name
        self.town = town

    # describes the class person
    def __str__(self):
```

```
        return self.first_name + ", " + self.last_name + ", " + self.town

    # falls back to this if there is no __str__ - used in debugging
    def __repr__(self):
        return self.first_name + ", " + self.last_name + ", " + self.town


    #CANNOT HAVE VARIABLE NAME SAME AS FUNCTION NAME

    @property
    def firstname(self):
        return self.first_name

    @firstname.setter
    def firstname(self, first_name):
        self.first_name = first_name

    @property
    def lastname(self):
        return self.last_name

    @lastname.setter
    def lastname(self, last_name):
        self.last_name = last_name

    @property
    def city(self):
        return self.town

    @city.setter
    def city(self, town):
        self.town = town

    # methods
    def full_name(self):
        """ Returns the full name (first and last name) as a string """
        return self.first_name + " " + self.last_name

    def full_name_city(self):
        """ Returns first, last and city as string """
        return self.first_name + " " + self.last_name + self.city

    def username(self):
        """ returns the username as string """
        return self.first_name[0] + self.last_name

# - - - - - - - - - - - - - - - - End Of Person - - - - - - - - - - - - - - - -

class Worker(Person):
    """ represents a worker """

    def __init__(self, first_name = "", last_name= "", town= "",
                 job_title= "", company_name= "", wages=0):
```

```
        self.job_title = job_title
        self.company_name = company_name
        self.wages = wages

        Person.__init__ (self, first_name, last_name, town)

    # describes the class person
    def __str__(self):
        return self.first_name + ", " + self.last_name + ", " + self.town
               + " "  + self.job_title + ", " + self.company_name +
               ", " +str(self.wages)

    # falls back to this if there is no __str__ - used in debugging
    def __repr__(self):
        return self.first_name + ", " + self.last_name + ", " + self.town
               + " "  + self.job_title + ", " + self.company_name +
               ", " +str(self.wages)

    @property
    def jobtitle(self):
        return self.job_title

    @jobtitle.setter
    def jobtitle(self, title_value):
        self.job_title = title_value

    @property
    def company(self):
        return self.company_name

    @company.setter
    def company(self, company_value):
        self.company_name = company_value

    @property
    def salary(self):
        return self.wages

    @salary.setter
    def salary(self, wages_value):
        self.wages = wages_value
# - - - - - - - - - - - - - - - - - End Of Worker - - - - - - - - - - - - - - - - -

class Voter(Person):
    """ Represents a voter based on Person """
    def __init__(self, first_name = "", last_name= "", town= "",
                 precient_number= 0, party_name= ""):
        self.precient_number = precient_number
        self.party_name = party_name

        Person.__init__ (self, first_name, last_name, town)

    def __str__(self):
```

```
        return self.first_name + " " + self.last_name + " " + self.town
           + " "  + str(self.precient_number) + " " + self.party_name

    # falls back to this if there is no __str__ - used in debugging
    def __repr__(self):
        return self.first_name + ", " + self.last_name + ", " + self.town
           + ", "  + str(self.precient_number) + ", " + self.party_name

    @property
    def precient(self):
        """represents the precient of the voter as a string """
        return self.precient_number

    @precient.setter
    def precient(self, precient_value):
        self.precient_number = precient_value

    @property
    def party(self):
        """represents the party of the voter as a string"""
        return self.party_name

    @party.setter
    def party(self, party_value):
        self.party_name = party_value

# - - - - - - - - - - - - - - - - - End Of Voter - - - - - - - - - - - - - - - -


def main():

    qty  = 5


    print("Create Person")
#    p1 = Person("John", "Doe", "Anywhere")

    p1 = Person("John","Doe","Kali")
    p1.firstname = "Jimbo"
    p1.lastname = "Smith"
    p1.city = "Whitefish"

    print(p1)
    print(p1.full_name())
    print(p1.first_name)
    print(p1.lastname)
    print(p1)

    p2 = Person("Sally", "Smith", "Polson")
    #change first and last name of p2
    p2.first_name = "Bonnie"
    p2.last_name = "Bond"
    print(p2)
```

```
# create a list of people
people = []

for i in range(qty):
    px = Person("John" + str(i),
                "Doe" + str(i),
                "Kali" + str(i))
    people.append(px)
    print(people[i].first_name + " "+ people[i].last_name)

print()

# create workers
w1 = Worker("Tom", "Tune", "New York", "Dancer", "Broadway", 100000)
print(w1)

w2 = Worker()
w2.firstname = "Ira"
w2.lastname = "Money"
w2.company = "IRS"
w2.jobtitle = "Auditor"
w2.salary = 90000


print(w2)

print()

# create a list of workers
workers = []



for i in range(qty):
    wx = Worker("John" + str(i),
                "Doe" + str(i),"",
                "Programmer" + str(i))
    workers.append(wx)
    print(workers[i].first_name + " "+
          workers[i].last_name  + " " +
          workers[i].jobtitle)

print()

# create voters
v1 = Voter("Abe", "Lincoln", "Springfield", 17, "Republican")
v2 = Voter()

v2.firstname = "Charles"
v2.lastname = "Daily"
v2.city = "Chicago"
v2.precient_number= 1
```

```
    v2.party ="Democrat"
    print(v1)
    print(v2)

    print()

    # create a list of voters
    voters = []

    for i in range(qty):
        vx = Voter("John" + str(i),
                   "Doe" + str(i),"","",
                   "Independent" + str(i))
        voters.append(vx)
        print(voters[i].firstname + " "+
              voters[i].lastname  + " " +
              voters[i].party)


    print("bye")

main()
```

Example 3 - getters and setters

```
# -* - coding: utf -8 -* -
"""
@author: jgoudy
"""
class class1:

    # The traditional "old" style of creating getters and setters

    def __init__(self, number = 0):
        self.number = number

    # getter method
    def get_Number(self):
        return self.number

    # setter method
    def set_Number(self, value):
        self.number = value

# - - - - - - - - - - - - - - - - - - End of class 2 - - - - - - - - - - - - - - - - - -

class class2:

    # using properties
    # using a property allows the object code to be cleaner
    # and users of the class will not have to make changes to their code

    def __init__(self, anumber = 0):
        self.anumber = anumber
```

```
    # getter method
    def get_Number(self):
        return self.anumber

    # setter method
    def set_Number(self, value):
        self.anumber = value

    # function to delete the number attribute
    def del_Number(self):
        del self.anumber

    # TURN SETTERS AND GETTERS INTO PROPERTIES
    number = property(get_Number, set_Number, del_Number, "Stores a number")

# - - - - - - - - - - - - - - - - - - End of class 2 - - - - - - - - - - - - - - - - - -

class class3:


    # THE PYTHONIC WAY
    # This is the preferred way of doing setters and getters.

    # this is a getter method

    def __init__(self, anumber = 0):
        self.anumber = anumber

    @property
    def number(self):
        return self.anumber

    # this is the setter method
    @number.setter
    def number(self, value):
        self.anumber = value

    @number.deleter
    def number(self):
        del self.anumber
# - - - - - - - - - - - - - - - - - - End of class 3 - - - - - - - - - - - - - - - - - -

def main():

    #Old way
    mc1 = class1()
    mc1.set_Number(42)
    print(mc1.get_Number())


    print("\n - - - - - - - - - -\n")
    mc2 = class2()
```

```
    mc2.number = 52
    print(mc2.number)


    print("\n - - - - - - - - - -\n")
    mc3 = class3()
    mc3.number = 62
    print(mc3.number)



main()
```

Example 4 - Trains

```
# trains
"""
Train
    id
    engine
    cabose
    start       (location)
    destination (location)
    traincars

"""
import random

class Train:

    # constructor
    def __init__(self, a_trainid, a_start="Whitefish",a_destination=""):
        self.a_trainid = a_trainid
        self.a_start = a_start
        self.a_destination = a_destination

        # create an empty list to hold train cars
        self.traincars = []

        # functions can be called within a constructor
        # or any other function as well
        self._addengine()

    # properties
    @property
    def trainid(self):
        return self.a_trainid
    @trainid.setter
    def trainid(self,value):
        self.a_trainid = value

    @property
    def start(self):
        return self.a_startstart
```

```
    @start.setter
    def start(self,value):
        self.a_start = value

    @property
    def destination(self):
        return self.a_destination
    @destination.setter
    def destination(self,value):
        self.a_destination = value


    # add our methods

    def _addengine(self):
        self.traincars.append("engine")

    def addcar(self,value):
        self.traincars.append(value)

    def addcaboose(self):
        self.traincars.append("caboose")

    # print the train cars
    def printtrain(self):
        print("\n"+ self.a_trainid + "\t" + self.a_start + " to " + self.a_destination)

        for i in range(len(self.traincars)):
            print("\t" + self.traincars[i])

        print()

# - - - - - - - - - - - - - - - - - - End Of Class - - - - - - - - - - - - - - - - - - - - - - - -

# ************************************************************
# program starts here

# global variable to hold whole trains
mytrains = []

# create a function to create a train with random cars
def createtrain(maxNumTrains, maxNumCars):

    # create a list car types
    cartypes = ["flat","box","tanker","passenger","ore"]

    # create a list of towns
    towns = ["Denver", "Seattle","Missoula", "Kalispell",
             "Whitefish", "Chicago", "New York", "Ringling","Clarskville"]

    # randomly pick a number to determine the number of trains
    rnd = random.randint(0,maxNumTrains)
```

```
    # loop to create the number of complete trains
    for i in range(rnd):

        # variables to hold the start and end towns
        startTown   = ""
        desTown     = ""

        # this while statement is used to ensure
        # that there will always be two different towns

        # if the startTown and desTown are equivalent,
        # the while loop runs. Note that the
        # startTown and desTown start equivalent

        while(startTown == desTown):
            # randomly going to pick a start town
            rndtemp = random.randint(0,len(towns) -1)
            startTown = towns[rndtemp]

            # randomly going to pick a destination town
            rndtemp = random.randint(0,len(towns) -1)
            desTown = towns[rndtemp]

        # call the class to create a train
        tx = Train("T"+ str(i),startTown,desTown)

        # randomly decide on how many train cars
        rndtemp = random.randint(0,maxNumCars)

        # randomly add cars to the train
        for i in range(rndtemp):
            rndcar= random.randint(0,len(cartypes) -1)
            tx.addcar(cartypes[rndcar])


        tx.addcaboose()

        # add train to the global variable
        mytrains.append(tx)


def printmytrains(alist):

    # print the total number of trains
    print(len(alist))

    # iterate/loop the number of trains
    for i in range(len(alist)):
        alist[i].printtrain()

        # this code will only show two trains at a time
        # if the remainder == 0 show the input command
        # example i = 4 | 4 / 2 = 0 with a remainder of 0
```

```
        if((i % 2) == 0):
            input("press any key to continue")



def main():

    # create an individual train
    train1 = Train("ML11","Helena","Missoula")

    # add 10 box cars
    for c  in range(10):
        train1.addcar("box")

    train1.addcaboose()

    # print the train we just created
    train1.printtrain()

    # create trains
    # each time this runs the trains will change
    createtrain(15,8)

    printmytrains(mytrains)


main()
```

0.19.3 How to Write a Class in Python

What is a class?

A **class** in Python is a blueprint for creating objects (a specific instance of a class). Classes allow us to encapsulate data (attributes) and functions (methods) that operate on that data. They help organize code in a structured way, making it reusable and easier to maintain.

When we say that *classes allow us to encapsulate data and functions that operate on that data*, we mean that a class groups together related pieces of information (called *attributes*) and the actions (called *methods*) that can be performed on that information. Encapsulation is a core concept in object-oriented programming (OOP) that helps in organizing and structuring code in a way that makes it more modular and manageable.

Let's break down these ideas:

1. **Attributes (Data)**: Attributes represent the properties or characteristics of an object created from a class. In our `Car` example, attributes like `make`, `model`, `year`, and `color` represent data about a specific car. These attributes are encapsulated within the class, meaning they are directly associated with the class and any objects made from it. For instance, if you create a `Car` object, you can think of `make`, `model`, etc., as data that specifically belongs to that object.

2. **Methods (Functions)**: Methods are functions defined inside a class that operate on its attributes or perform actions related to the class. For instance, methods like `start` and `stop` are actions that a `Car` object can perform. These methods may use or change the class's attributes, creating a relationship between the data and the actions associated with that data.

3. **Encapsulation**: Encapsulation is the idea that all related attributes and methods are "wrapped up" within the class, forming a distinct, self-contained unit. This provides several benefits:

   - **Organization**: All relevant data and functions for a specific concept (e.g., a car) are organized within a single class.
   - **Data Protection**: By keeping data within a class, we can control how it's accessed and modified. For instance, using *getter* and *setter* methods lets us add validation or restrictions.
   - **Reusability and Modularity**: Once a class is defined, it can be used as a standalone component. If we want to create multiple `Car` objects, each object will have its own data and can operate independently of others.

In short, encapsulation helps to bundle related information and functions together in a way that protects the data and keeps code organized, readable, and reusable.

We will create a class called `Car` that represents a car with attributes such as `make`, `model`, `year`, and `color`. We will also add methods to work with these attributes. Finally, we will test our class using the `if __name__ == "__main__"` method.

---

1. Declare the Class

To create a class, use the `class` keyword followed by the name of the class.

```
class Car:
    """A class to represent a car."""
```

2. Declare the Attribute Variables

Attributes are the characteristics of the class. Here, `make`, `model`, `year`, and `color` are attributes that define each car.

```
class Car:
    """A class to represent a car."""

    def __init__(self, make, model, year, color):
        self._make = make
        self._model = model
        self._year = year
        self._color = color
```

3. Create the Constructor

A **constructor** in Python is a special method called `__init__`. It is automatically called when a new object of the class is created. The purpose of the constructor is to initialize the object's attributes.

```
def __init__(self, make, model, year, color):
    """Initialize the car's attributes."""
    self._make = make
    self._model = model
    self._year = year
    self._color = color
```

4. Create `__str__`

The `__str__` method is a special method that returns a string representation of the object. This is useful when you want a readable output for instances of the class.

```
def __str__(self):
    """Return a string representation of the car."""
    return f"{self._year} {self._make} {self._model} in {self._color} color"
```

5. Create `__repr__`

The `__repr__` method is used for a developer-oriented string representation of an object. While similar to `__str__`, `__repr__` is generally intended for debugging.

```
def __repr__(self):
    """Return a detailed string representation for debugging."""
    return f"Car(make={self._make}, model={self._model}, year={self._year}, color={self._color
```

6. Create Properties

Properties provide controlled access to attributes by defining getter and setter methods. This is useful for validating or processing data before assigning it to an attribute.

```
@property
def make(self):
    """Get the make of the car."""
    return self._make

@make.setter
def make(self, value):
    """Set the make of the car."""
    self._make = value

@property
def model(self):
```

```
    """Get the model of the car."""
    return self._model

@model.setter
def model(self, value):
    """Set the model of the car."""
    self._model = value
```

7. Create Methods

Methods define actions or behaviors for objects of the class. Here, we define two methods: `start` and `stop`.

```
def start(self):
    """Start the car."""
    return f"{self._make} {self._model} has started."

def stop(self):
    """Stop the car."""
    return f"{self._make} {self._model} has stopped."
```

Full Code with Testing

Here's the full code, including the class definition, properties, methods, and a test block.

```
class Car:
    """A class to represent a car."""

    # - - - - - - - - - - - - CONSTRUCTOR - - - - - - - - - - - -
    def __init__(self, make, model, year, color):
        """Initialize the car's attributes."""
        self._make = make
        self._model = model
        self._year = year
        self._color = color

    # - - - - - - - - - - - - DEBUGGING TOOLS - - - - - - - - - - - -
    def __str__(self):
        """Return a string representation of the car."""
        return f"{self._year} {self._make} {self._model} in {self._color} color"

    def __repr__(self):
        """Return a detailed string representation for debugging."""
        return f"Car(make={self._make}, model={self._model}, year={self._year}, color={self._c

    # # - - - - - - - - - - - - PROPERTIES  - - - - - - - - - - - -
    @property
    def make(self):
        """Get the make of the car."""
        return self._make

    @make.setter
    def make(self, value):
        """Set the make of the car."""
        self._make = value
```

```
    @property
    def model(self):
        """Get the model of the car."""
        return self._model

    @model.setter
    def model(self, value):
        """Set the model of the car."""
        self._model = value

    # - - - - - - - - - - - - METHODS - - - - - - - - - - - -
    def start(self):
        """Start the car."""
        return f"{self._make} {self._model} has started."

    def stop(self):
        """Stop the car."""
        return f"{self._make} {self._model} has stopped."


# Testing the Car class
if __name__ == "__main__":
    my_car = Car("Toyota", "Camry", 2022, "blue")
    print(my_car)                                          # Output the car's information
    print(f"My car is a {my_car.make} {my_car.model}")  # Using the car's properties
    print(my_car.start())                                  # Start the car
    print(my_car.stop())                                   # Stop the car
    my_car.make = "Honda"                                  # Changing the car's make
    print(my_car)                                          # Output the car's information after c


'''
OUTPUT
2022 Toyota Camry in blue color
Toyota Camry has started.
Toyota Camry has stopped.
2022 Honda Camry in blue color
'''
```

Summary

In this guide, we explored how to write a class in Python. We learned:

1. **Class Declaration**: Using the `class` keyword to create a new class.
2. **Constructor**: Initializing an object's attributes with `__init__`.
3. **Special Methods**: Using `__str__` for readable string representation and `__repr__` for debugging output.
4. **Properties**: Using getter and setter methods to manage access to private attributes.
5. **Methods**: Defining actions for the class, like `start` and `stop`.
6. **Testing the Class**: Verifying functionality using `if __name__ == "__main__"`.

This framework enables us to build and interact with custom objects in a structured way.

0.19.4 Decorators

Decorators are:

A design pattern that allows you to modify the behavior of a function or method without directly changing its code. They are essentially functions that wrap other functions, adding extra functionality before, after, or around the original function's execution.

How they work:

Decorator function:

- Takes another function as an argument.
- Returns a modified version of the original function.

Applying a decorator:

- Use the ''@'decorator_name syntax above a function definition.
- This is shorthand for function_name = decorator_name(function_name).

Remember:

- Decorators are functions that take another function as an argument and return a modified version of that function.
- They provide a powerful way to add functionality to existing functions without modifying their code directly.
- Use them to enhance code readability, maintainability, and reusability.

(AIJG)

```
# Decorator Example 1

# a list to hold the log entries
listOfFunctionCalls = []

# log every function call
# in testing etc, this easily could be written
# to a log file

def log_function_call(func):
    def wrapper(*args):
        result = func(*args)
        # print(f"Function '{func.__name__}' called with arguments {args} and {kwargs}. Return
        xFunCall = (f"Function '{func.__name__}' called with arguments {args}. Returned {resul
        listOfFunctionCalls.append(xFunCall)
        return result
    return wrapper

@log_function_call
def add(x, y):
    return x + y

@log_function_call
def sub(x,y):
    return x - y

@log_function_call
def multiply(x,y):
    return x * y

def printFunctionsCalled():
```

```
    print("A list of functions called")
    for i in listOfFunctionCalls:
        print(i)

def main():

    print(add(5, 3))
    print(sub(10,6))
    print(multiply(25,4))

    print('\n')

    printFunctionsCalled()
main()

"""
Output

8
4
100


A list of functions called
Function 'add' called with arguments (5, 3). Returned 8
Function 'sub' called with arguments (10, 6). Returned 4
Function 'multiply' called with arguments (25, 4). Returned 100
"""
```

Example2

```
import time



listOfFunctionCalls = []


def timeit(func):
    """Decorator to time the execution of a function."""

    def wrapper(*args, **kwargs):
        start_time = time.time_ns()
        result = func(*args, **kwargs)
        end_time = time.time_ns()
        timedef = (f"Function '{func.__name__}' took {(end_time - start_time)} nano sec to exe
        listOfFunctionCalls.append(timedef)
        return result

    return wrapper

def printFunctionsCalled():

    print("A list of functions called")
```

```
    for i in listOfFunctionCalls:
        print(i)
    print()


@timeit
def loopcount1(n):

    a = 0

    for x in range(n):
        a = a + x

    print(a)

@timeit
def loopcount2(n):

    a = 0

    for x in range(0,n,50):
        a = a + (x)
    print(a)

def main():

    x = 1000000

    loopcount1(x)
    loopcount2(x)

    printFunctionsCalled()


main()

"""
499999500000
9999500000
A list of functions called
Function 'loopcount1' took 38899200 nano sec to execute.
Function 'loopcount2' took 1033200 nano sec to execute.
"""
```

0.19.5 Properties

A Python property is a special construct that allows you to control access to an object attribute through methods, while still using dot notation as if it were a regular attribute. This means you can perform additional logic, validation, or computations when getting or setting the attribute's value.

Key points:

- **Methods disguised as attributes:** Properties are essentially methods that act like attributes, making code more readable and intuitive.
- **Encapsulation:** They promote encapsulation by hiding the implementation details of how an attribute is accessed or modified.
- **Validation and control:** You can add validation or constraints to ensure data integrity, as well as perform side effects when the attribute is accessed or changed.
- **Readability:** They make code cleaner and more maintainable by avoiding explicit getter and setter methods.

(AIJG)

Lecture Code

```
class Person:
   """Represents a person with a first and last name."""

   def __init__(self, first_name: str, last_name: str) -> None:
       """Initializes a Person object with the given first and last name."""
       self._first_name = first_name
       self._last_name = last_name

   @property
   def full_name(self) -> str:
       """Returns the full name of the person."""
       return f"{self._first_name} {self._last_name}"

   @property
   def email(self) -> str:
       """Returns the email address of the person."""
       # Replace with actual logic
       return f"{self._first_name}.{self._last_name}@example.com"


   # - - - - - - - - - - - - - - - - - - - - Another Class - - - - - - - - - - - - - - - - - - -
# lazy loading

class Book:
    def __init__(self, title="", author=""):
        self.atitle = title
        self.aauthor = author

    # REMEMBER THAT PROPERTY NAMES NEED TO BE UNIQUE
    @property
    def bookTitle(self):
        return self.atitle
    @bookTitle.setter
    def bookTitle(self,title):
        self.atitle = title
```

```
    @property
    def bookAuthor(self):
        return self.aauthor
    @bookAuthor.setter
    def bookAuthor(self,author):
        self.aauthor = author

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def main():

    print("\nPeson Property Example")

    # create an new person object
    p1 = Person("Bubba","Doe")

    print("Name: {} /nEmail: {}\n\n".format(p1.full_name,p1.email))

    print("\nBook Property Example")

    # create new book object
    b1 = Book("Code Gurus Unleashed")
    b1.bookAuthor = "Jimbo"

    print("Title: {}".format(b1.bookTitle))
    print("Author: {}".format(b1.bookAuthor))

    print("\nbye\n")

main()

# Output
"""
Peson Property Example
Name: Bubba Doe /nEmail: Bubba.Doe@example.com

Book Property Example
Title: Code Gurus Unleashed
Author: Jimbo

bye
"""
```

0.19.6 When to use Setters and Getters, Decorators, and Properties

Decorators:

- **Definition:** Functions that modify the behavior of other functions or classes.
- **Syntax:** `@decorator_name` above the function or class definition.
- **Purpose**:
  - Add functionality before, after, or around function calls.
  - Manage access to attributes.
  - Enforce validation or restrictions.
  - Implement caching or logging.
  - Simplify code by automating repetitive tasks.

Getters and Setters (Traditional Approach):

- **Definition:** Methods that control access to class attributes.
- **Syntax:** Explicit method calls (e.g., `object.get_attribute()`, `object.set_attribute(value)`).
- **Purpose**:
  - Encapsulate data and protect its integrity.
  - Perform validation or transformation on attribute values.
  - Trigger side effects when attributes are accessed or modified.

Properties (Pythonic Approach using Decorators):

- **Definition:** A way to create getters and setters using the `@property` decorator.
- **Syntax:** `@property` for the getter, `@attribute.setter` for the setter.
- **Advantages**:
  - Cleaner syntax for accessing and modifying attributes (like regular attributes).
  - Encapsulation and control over attribute access.
  - Custom logic within getters and setters.

Comparison Table:

Feature	Decorators	Getters/Setters (Traditional)	Properties
Syntax	`@decorator`	Explicit method calls	`@property`
Purpose	Modify functions/classes	Control attribute access	Getters/setters using decorators
Flexibility	Broader range of applications	Specific to attribute access	Focus on attribute access
Readability	Depends on decorator complexity	Clearer separation of concerns	Cleaner attribute access
Pythonic style	More Pythonic	Traditional approach	Pythonic approach

When to Use Which:

- **Decorators:** For general-purpose function and class modification, not limited to attribute access.
- **Properties:** For cleaner attribute access with encapsulation and custom logic.
- **Traditional getters/setters:** For explicit method calls or specific coding conventions.

In general, properties are the preferred approach in Python due to their readability and Pythonic style. However, decorators offer more flexibility for broader use cases beyond attribute management.

0.19.7 Pet Search Program — Explanation and Learning Outcomes

Program Overview

This Python program demonstrates **object-oriented programming (OOP)** through the creation and management of a `Pet` class. Each `Pet` object stores three key attributes — **name, breed, and type** — and the program builds a list of these objects to simulate a simple pet database. It then performs three main tasks:

1. **Displays all pets** in a neatly formatted list.
2. **Searches for a pet by name**, using case-insensitive comparison.
3. **Counts and reports the number of dogs and cats** in the dataset.

Through this, students gain hands-on experience with **class construction, object instantiation, encapsulation, list manipulation, and search algorithms** — all core principles in introductory computer science.

Key Concepts Demonstrated

1. **Class Design and Object-Oriented Thinking**
   - Introduces the idea of modeling real-world entities (pets) as **objects**.
   - Demonstrates **constructors (`__init__`)**, **instance variables**, and **string representations (`__str__`)**.
2. **Encapsulation and Data Access**
   - Reinforces **data hiding** and controlled access through Python's `@property` and setter decorators.
   - Encourages writing robust, maintainable code where attributes are accessed safely.
3. **List-Based Data Storage**
   - Demonstrates how to store and traverse lists of objects.
   - Highlights how lists serve as flexible containers for structured data.
4. **Functional Decomposition**
   - Separates logic into reusable functions (`addPetsToList`, `findPetName`, `countTypes`, and `main`).
   - Emphasizes **top-down design** and the importance of a clean, organized program structure.
5. **Search Logic and Conditional Flow**
   - Implements a **case-insensitive search algorithm**.
   - Uses `if` statements and return values to control flow and handle edge cases.
6. **Iteration and Aggregation**
   - Demonstrates looping through objects to perform operations such as counting and filtering.
   - Reinforces accumulation patterns (`dogCount`, `catCount`) in programming logic.
7. **Debugging and Output Formatting**
   - Encourages readable console output and diagnostic print statements.
   - Reinforces understanding of formatted strings (f-strings) for expressive output.

8. **Code Readability and Documentation**
    - Uses clear variable names, meaningful comments, and modular design.
    - Promotes industry-aligned coding habits and Python style conventions (PEP 8).

Example Code

```
# ------------------------------------------------------------
# Program: Pet Search Example (Enhanced)
# Author: Jim Goudy
# Description:
#   Demonstrates creating a Pet class, storing Pet objects,
#   searching for pets by name (case -insensitive) and
#   counting the number of cats and dogs.
#
#   The program lists pets, searches by name, and reports
#   the exact match if found.
# ------------------------------------------------------------


# ------------------------------------------------------------
# Class Definition: Pet
# ------------------------------------------------------------
class Pet:

    def __init__(self, a_petname="", a_petbreed="", a_pettype=""):
        """Constructor initializes the pet's name and breed."""
        self.a_petname = a_petname
        self.a_petbreed = a_petbreed
        self.a_pettype = a_pettype

    def __str__(self):
        """Returns a readable string representation of the pet."""
        return f"{self.a_petname} ({self.a_petbreed} {self.a_pettype})"

    # Property methods for pet name
    @property
    def petname(self):
        """Gets the pet's name."""
        return self.a_petname

    @petname.setter
    def petname(self, value):
        """Sets the pet's name."""
        self.a_petname = value

    # Property methods for pet breed
    @property
    def petbreed(self):
        """Gets the pet's breed."""
        return self.a_petbreed
```

```
    @petbreed.setter
    def petbreed(self, value):
        """Sets the pet's breed."""
        self.a_petbreed = value

    # Property methods for pet type
    @property
    def pettype(self):
        """Gets the pet's type."""
        return self.a_pettype

    @pettype.setter
    def pettype(self, value):
        """Sets the pet's breed."""
        self.a_pettype = value

    # - - - - - - - - - - - - - - Methods - - - - - - - - - - - - - - - - - - - - - -

    def printPet(self):
        """Prints the pet's name and breed in a single line."""
        print(f"{self.petname} - {self.petbreed}")


# - - - - - - - - - - - - - - - - - - End Of Class - - - - - - - - - - - - - - - - - -


# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Global Variables and Functions
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# List to store all Pet objects
listOfPets = []


def addPetsToList(nameList, breedList,types):
    """
    Creates Pet objects from name and breed lists
    and appends them to the global listOfPets.
    """
    for i in range(len(nameList)):
        px = Pet(nameList[i], breedList[i],types[i])
        listOfPets.append(px)
        px.printPet()


def findPetName(findName):
    """
    Searches the global list for a pet with the given name.
    Comparison is case -insensitive.
    Returns the Pet object if found, otherwise None.
    """
    for apet in listOfPets:
```

```
        if findName.lower() == apet.petname.lower():  # case -insensitive match
            return apet
    return None

def countTypes(alist):
    """
    This function counts how many dogs and cats are in a
    list of pet objects. It starts by initializing two counters
    dogCount and catCount  both set to zero. Then, it loops through
    each apet in listOfPets (note: the function parameter alist is unused).

    For each pet, it prints its type (apet.pettype) and checks if the
    pets type is "dog". If it is, it increments the dog counter. Otherwise,
    it assumes the pet is a cat and increments the cat counter.
    """

    dogCount = 0
    catCount = 0

    for apet in listOfPets:
        print("ttt: " + apet.pettype)
        if(apet.pettype.lower() == "dog" ):
            dogCount +=1
        else:
            catCount = catCount + 1

    print(f"Dog count = {dogCount}")
    print(f"Cat count = {catCount}")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Main Program
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def main():
    """Main driver function for pet creation and search."""

    # 15 pet names (dogs and cats mixed)
    petnames = [
        "Buddy", "Luna", "Charlie", "Bella", "Max",
        "Milo", "Lucy", "Rocky", "Daisy", "Leo",
        "Chloe", "Cooper", "Nala", "Simba", "Zoe"
    ]

    # 15 breeds (dogs and cats mixed together)
    breeds = [
        "Labrador Retriever", "Beagle", "German Shepherd", "Poodle", "Bulldog",
        "Golden Retriever", "Siamese", "Persian", "Mutt", "Sphynx",
        "Bengal", "Ragdoll", "Shih Tzu", "Boxer", "Chihuahua"
    ]

    types = [
        "dog", "dog", "dog", "dog", "dog",
        "dog", "cat", "cat", "dog", "cat",
        "dog", "cat", "dog", "dog", "dog"
```

```
    ]

    # Populate list of pets
    print("Pet List:")
    addPetsToList(petnames, breeds,types)

    print("\n")  # Blank line for readability

    # Search for a specific pet name
    searchName = "luCy"  # lower case for demo
    result = findPetName(searchName)

    if result:
        print(f"Pet found: {result.petname} is a {result.petbreed}.")
    else:
        print(f"Pet '{searchName}' was NOT found.")


    countTypes(listOfPets)

    print("\nbye\n")


# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Program Entry Point
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
main()
```

0.20 NumPy / Arrays

0.20.1 What is an array?

An array is a data structure that stores a collection of items of the same data type and usually stores them in a continuous memory location. Each item in an array is assigned a unique index, which is used to access and manipulate the item. Arrays are a fundamental data structure in computer science and are used in a wide variety of applications, such as sorting, searching, and data processing.

0.20.2 What is NumPy?

NumPy stands for Numerical Python. It is the Python library that is used for working with arrays.

0.20.3 Array Examples

```
"""
Programmer: James Goudy

This code shows examples of
multi -dimesional arrays

"""

import numpy as np
import sys


# Global Variables

# one dimensional array
arr1 = np.array([1, 2, 3])


# two dimensional array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# three dimensional array
arr3 = np.array([[[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]],
                 [[10, 11, 12],
                  [13, 14, 15],
                  [16, 17, 18]]])

# four dimensional array
arr4 = np.array([[[[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]],
                  [[10, 11, 12],
                   [13, 14, 15],
                   [16, 17, 18]],
                  [[19, 20, 21],
                   [22, 23, 24],
                   [25, 26, 27]]],
                 [[[28, 29, 30],
```

```
                    [31, 32, 33],
                    [34, 35, 36]],
                   [[37, 38, 39],
                    [40, 41, 42],
                    [43, 44, 45]],
                   [[46, 47, 48],
                    [49, 50, 51],
                    [52, 53, 54]]])


def one_dimension():

    # one dimensional array

    print("\n1 Dimensional - Output")

    # print the shape
    print(str(arr1.shape))

    # Iterate through each box
    for c in range(3):

        sys.stdout.write(str(arr1[c]) + " ")

    print("\n")


def two_dimension():

    # two dimensional array
    print("\n2 Dimensional - Output")

    # print the shape
    print(str(arr2.shape))

    # Iterate through each box
    for r in range(2):
        for c in range(3):

            sys.stdout.write(str(arr2[r][c]) + " ")

        print()

    print("\n")


def three_dimesion():

    # three dimensional array
    print("\n3 Dimensional - Output")

    # print the shape
    print(str(arr3.shape))
```

```
    # Iterate through each box
    for dd in range(2):

        for r in range(3):

            for c in range(3):

                sys.stdout.write(str(arr3[dd][r][c]) + " ")

    print("\n")


def fourth_dimension():

    # fourth dimensional array

    print("\n4 Dimensional - Output")

    print("Shape " + str(arr4.shape) + "\n")

    # iterate through each box
    for ff in range(2):
        for tt in range(3):
            for r in range(3):
                for c in range(3):

                    sys.stdout.write(str(arr4[ff][tt][r][c]) + " ")

    print("\n")


def main():

    one_dimension()

    two_dimension()

    # note three and four dimensioanl arrays
    # are being printed in a linear fashion

    three_dimesion()

    fourth_dimension()


main()

"""
1 Dimensional - Output
(3,)
1 2 3
```

```
2 Dimensional - Output
(2, 3)
1 2 3
4 5 6



3 Dimensional - Output
(2, 3, 3)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18


4 Dimensional - Output
Shape (2, 3, 3, 3)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 3
"""
```

0.20.4 NumPy Functions

```
# -* - coding: utf -8 -* -
"""
Created on Fri Sep 29 11:06:03 2023
Programmer: James Goudy
Title: Programming Guru

@author: jgoudy

"""

# note that we are assigning np as an alias/handle
import numpy as np
import sys


def printArray(theArray, rows, cols):

    maxStringLen = 0

    # the numpy function max() cannot work on a string array
    # in order to find the largest string, one has to iterate
    # through each individual item.  len() can be used on a string.
    # So each item is checked and the largest is kept in the
    # variable maxStringLen

    # the code checks to see if the array is a string or numbers
    # if it is a string ('<U8') then the strings are iterated one
    # by one checking the length and recording the largest

    # else - the array is a number, in which the numpy.max() and numpy.mni()
    # function will work and it will find the largest smallest value.
    # However, the absolute value needs to be used to take in account if
    # it is positive or negative.
```

```
    if theArray.dtype == '<U8':
        for r in range(rows):
            for c in range(cols):
                ilen = len(theArray[r][c])
                if (ilen > maxStringLen):
                    maxStringLen = ilen

        maxStringLen += 2

        if rows == 1:

            for c in range(cols):
                sys.stdout.write(str(theArray[c]).rjust(maxStringLen, " "))

        else:

            for r in range(rows):
                for c in range(cols):
                    sys.stdout.write(theArray[r][c].rjust(maxStringLen, " "))
                print()

    else:

        # save max and min lengths account for the sign
        # for instance -320 has greater length than 75
        # abs -> str so the len() function can be used

        maxVal = len(str(abs(theArray.max())))
        minVal = len(str(abs(theArray.min())))

        maxValLen = 0

        # ternary operator in python
        maxValLen = maxVal if maxVal > minVal else minVal

        # add to to the separation
        maxValLen += 2

        # get the max value and turn it into a string
        # value = str(theArray.max())

        # get the length of the string and add 1
        # maxLen = len(value)+ 2

        if rows == 1:

            for c in range(cols):
                sys.stdout.write(str(theArray[c]).rjust(maxValLen, " "))

        else:

            for r in range(rows):
```

```
                for c in range(cols):
                    sys.stdout.write(str(theArray[r][c]).rjust(maxValLen, " "))
                print()

    print("\n - - - - - - - - - - - - - - - -\n")


def numpydemo():

    # create a two dimensional array
    arr = np.array([[11, 12, 13, 14],
                    [21, 22, 23, 24],
                    [31, 32, 33, 34]])

    rows = 3
    cols = 4

    # print 2 dimensional array

    # this for statement keeps track of the row
    for row in range(rows):

        # this for statement keeps track of the col
        for col in range(cols):
            sys.stdout.write(str(arr[row, col]) + " ")

        # When inner forstatement is done we are at the end of the row
        print()

    print("\n")

    # print an individual element arrayname[row,column]
    print("second row, third column " + str(arr[1, 2]))
    print("\n")

    # number of dimensions .ndim
    print("Number of dimensions = " + str(arr.ndim))
    print("\n")

    # shape of the array .shape
    print("Shape of array " + str(arr.shape))
    print("\n")

    # print the length of the array len(arrayname)
    # note that length does not take in account the rows
    print("The length is " + str(len(arr)))
    print()

    # print the sum of the array .sum()
    print("The sum of the array is " + str(arr.sum()))
    print()

    # print the max value of the array .max()
```

```
print("The max of the array is " + str(arr.max()))
print()

# print the min value of the array .min()
print("The min of the array is " + str(arr.min()))
print()

# print the mean value of the array .mean()
print("The mean of the array is " + str(arr.mean()))
print()

# print the standard deviation value of the array .std()
print("The standard deviation of the array is " + str(arr.std()))
print()

# print add row 1 column 2 value with
# row 2 column 2 value
sum = arr[0, 1] + arr[1, 1]
print("The sum of row 1 column 2 \n" +
      "with row 2 column 2 is " + str(sum))

# create a 2 dimensional array that is filled with random numbers
rows = 5
cols = 6

print("\nArray with random integers")
arr55 = np.arange(rows*cols).reshape(rows, cols)
print("\narr55")
printArray(arr55, rows, cols)

# create an empty array filled with zeros
arr66 = np.zeros((rows, cols))
print("arr66 empty array with zeros")
# print(arr66)
printArray(arr66, rows, cols)

# String array
print("\nStates and Capitals")
arr77 = np.array([["Helena", "Albany", "Columbus"],
                  ["MT", "NY", "OH"]], dtype=np.dtype('U'))

printArray(arr77, 2, 3)

# this creates a one dimensional array
# with 16 elements filled with  random ints form 0 to 24
arr2 = np.random.randint(0, 25, 16)

print("\nArr2\n")
for i in arr2:
    sys.stdout.write(str(i) + " ")

print("\n")
```

```
    # convert the one dimensional array in to a 4 x 4
    arr2.resize(4, 4)

    rows = 4
    cols = 4

    printArray(arr2, rows, cols)

    print("\n")

    # boolean indexing
    print("Print values that are greater than 10")
    print(arr2[arr2 > 10])

    # Generate random letters - 65 -90 A -Z
    arr3 = np.random.randint(65, 90, 15)

    # generate a blank array
    arr4 = np.zeros(len(arr3), dtype=np.str_)
    # arr4 = np.empty(15, dtype = np.unicode)

    print("\nBlank Array arr4")
    print(arr4)
    print("\nPrint the random ascii codes in arr3")
    printArray(arr3, 1, 15)

    # populate the empty arr4 with the characters
    # from arr3
    for i in range(len(arr3)):
        arr4[i] = chr(arr3[i])

    print(arr4)
    print()

    # for i in range(len(arr4) -1):
    #     sys.stdout.write(str(arr4[i]) + " ")

    # print()

    # default is empty
    arr44 = np.empty([4, 4], dtype=np.int16)
    print(arr44)

    print()
    arr44.fill(42)
    print(arr44)


def main():

    numpydemo()
```

```
main()


"""
11 12 13 14
21 22 23 24
31 32 33 34


second row, third column 23


Number of dimensions = 2


Shape of array (3, 4)


The length is 3

The sum of the array is 270

The max of the array is 34

The min of the array is 11

The mean of the array is 22.5

The standard deviation of the array is 8.241156876717412

The sum of row 1 column 2
with row 2 column 2 is 34

Array with random integers

arr55
   0   1   2   3   4   5
   6   7   8   9  10  11
  12  13  14  15  16  17
  18  19  20  21  22  23
  24  25  26  27  28  29

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

arr66 empty array with zeros
  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  0.0
  0.0  0.0  0.0  0.0  0.0  0.0

 - - - - - - - - - - - - - - - -
```

```
States and Capitals
    Helena    Albany  Columbus
        MT        NY        OH

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


Arr2

15 3 7 20 2 2 15 13 10 9 9 4 16 8 5 24

  15   3   7  20
   2   2  15  13
  10   9   9   4
  16   8   5  24

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



Print values that are greater than 10
[15 20 15 13 16 24]

Blank Array arr4
['' '' '' '' '' '' '' '' '' '' '' '' '' '' '' '']

Print the random ascii codes in arr3
  68  74  84  85  89  83  82  89  87  73  70  88  78  70  89
 - - - - - - - - - - - - - - - -

['D' 'J' 'T' 'U' 'Y' 'S' 'R' 'Y' 'W' 'I' 'F' 'X' 'N' 'F' 'Y']

[[42 42 42 42]
 [42 42 42 42]
 [42 42 42 42]
 [42 42 42 42]]

[[42 42 42 42]
 [42 42 42 42]
 [42 42 42 42]
 [42 42 42 42]]
"""
```

0.21 Pandas

- **Pandas** is a Python library that provides fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It is built on top of the NumPy library and is particularly well-suited for working with tabular data, such as spreadsheets or SQL tables.
- Pandas introduces two new data structures to Python - **Series** and **DataFrame** - both of which are built on top of NumPy.
- **Pandas DataFrames** are two-dimensional, size-mutable, tabular data structures with labeled axes (rows and columns) 1. They are built on top of the NumPy library and are designed to handle a wide range of data manipulation tasks, such as filtering, reshaping, merging, and pivoting data 1.
  A **DataFrame** is composed of three main components: the **data**, the **index**, and **columns**. The data is stored in a two-dimensional array-like structure, while the index is an array of labels that represent the rows of the **DataFrame**. The columns are an array of labels that represent the columns of the **DataFrame**.
- **Pandas Series** is a one-dimensional labeled array-like object that can hold data of any type. A Pandas **Series** consists of two main components: the **data** and the **index**. The **data** is stored in a one-dimensional array-like structure, while the index is an array of labels that represent the rows of the Series

---

0.21.1 Common Panda Functions

Pandas are a powerful library that provides a wide range of functions for data manipulation in Python. Here are some of the most commonly used functions:

- **read_csv**(): Load data from a CSV file.
- **fillna**(): Replace missing values in a DataFrame.
- **max**(): Calculate the max of a Series or DataFrame.
- **mean**(): Calculate the mean of a Series or DataFrame.
- **min**(): Calculate the min of a Series or DataFrame.
- **std**(): Calculate the standard deviation of a Series or DataFrame.
- **value_counts**(): Count unique values in a Series.
- **groupby**(): Group DataFrame or Series using a mapper or by a Series of columns.
- **loc**[]: Access a group of rows and columns by label(s) or a Boolean array.
- **iloc**[]: Access a group of rows and columns by integer position(s).
- **unique**(): Return unique values in a Series.
- **nunique**(): Return the number of unique elements in a Series.
- **cut**(): Bin values into discrete intervals.
- **qcut**(): Quantile-based discretization function .
- **merge**(): Merge DataFrame or named Series objects with a database-style join.
- **concat**(): Concatenate pandas objects along a particular axis.

0.21.2 loc and iloc

In pandas, `loc` and `iloc` are two crucial methods for accessing and manipulating data in DataFrames. While they seemingly achieve similar tasks, they differ in their approach and have distinct strengths and weaknesses. Here's a breakdown of their key differences:

Indexing method:

- **loc:** Uses labels (row/column names) to select data. This is intuitive and easy to understand, especially for data exploration and analysis.
- **iloc:** Uses integer positions (indices) to select data. This is more efficient for specific indexing tasks like slicing or iterating through rows/columns.

Error handling:

- **loc:** Raises a `KeyError` if the label doesn't exist in the DataFrame. This can be helpful for catching errors in data manipulation.
- **iloc:** Raises an `IndexError` if the integer position doesn't exist in the DataFrame. This can be silent if you're not expecting a specific index, leading to unexpected results.

Adding/modifying data:

- **loc:** Can add new rows/columns based on labels, even if they don't exist initially. This makes it flexible for data manipulation.
- **iloc:** Can only modify existing data based on integer positions. It cannot add new rows/columns directly.

Here's a table summarizing the key differences:

Feature	loc	iloc
Indexing method	Labels (row/column names)	Integer positions (indices)
Error handling	Key error	Index error
Adding/modifying data	Can add new rows/columns	Can only modify existing data
Best for	Data exploration, analysis	Specific indexing tasks, slicing, iteration

Choosing the right method depends on your specific needs:

- Use **loc** when:
  - You want to select data based on labels/names for easier understanding.
  - You need to add new rows/columns to your DataFrame.
  - You want to catch errors related to invalid labels.
- Use **iloc** when:
  - You need efficient indexing based on specific positions.
  - You are slicing or iterating through rows/columns.
  - You know the exact positions of the data you need.

---

idxmax()

In pandas, `idxmax()` is a method used to find the index of the maximum value in a DataFrame or Series. It comes in handy for various data analysis tasks involving finding the top values or entries in your data.

Here's a breakdown of what `idxmax()` does:

- **Returns**: It returns a Series containing the index labels for the maximum values in each specified axis.
- **Axis:** By default, it searches for the maximum values **across each column** (axis=0). You can specify `axis=1` to find the maximum for each **row** instead.
- **NA values:** `idxmax()` automatically excludes null/NA values when searching for the maximum.

idxmin()

In pandas, `idxmin()` is the twin brother of `idxmax()`, but instead of finding the maximum values, it finds the **minimum values** in your data. It's another handy tool for data analysis tasks involving identifying the bottom values or entries.

Here's what you need to know about `idxmin()`:

- **Returns**: Similar to `idxmax()`, it returns a Series containing the **index labels** for the minimum values in each specified axis.
- **Axis:** By default, it searches for the minimums **across each column** (axis=0). You can specify `axis=1` to find the minimum for each **row** instead.
- **NA values:** Just like its partner, `idxmin()` automatically excludes null/NA values when searching for the minimum.

0.21.3 Panel Data - Pandas

The name **Pandas** in Python is derived from the term **"Panel Data"**, which refers to multidimensional structured datasets commonly used in statistics and econometrics. It is not strictly an acronym but rather a nod to the library's functionality for handling such data.

The official definition and purpose of Pandas in Python can be summarized as follows:

- **Definition:** Pandas is an open-source data manipulation and analysis library for Python, providing data structures and operations for manipulating numerical tables, time series, and other structured data formats.

While "Pandas" is often written as though it were an acronym, it is more of a convenient and memorable name linked to its ability to work with **panel data**. It also aligns with the library's playful and accessible branding, making it one of the most recognizable Python libraries.

Python's **Pandas** library provides a vast array of functions for data manipulation and analysis. Below are some of the **main functions** commonly used in Pandas, organized by their purpose:

Data Creation in Pandas

1. `pd.Series()` The `pd.Series()` function in Pandas is used to create a **one-dimensional labeled array** that can hold data of any type (e.g., integers, floats, strings, etc.). A **Series** is similar to a column in a table or a list in Python, but with **indexing** capabilities.

Key Points:

- A Series has two components:
  - **Data**: The values you want to store (e.g., numbers, strings, etc.).
  - **Index**: The labels for each data point (default is integer-based, starting from 0).

Example:

```
import pandas as pd

# Creating a Series from a list
s = pd.Series([10, 20, 30, 40])
print(s)
```

Output:

```
0    10
1    20
2    30
3    40
dtype: int64
```

Specifying Custom Index:

```
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print(s)
```

Output:

```
a    10
b    20
c    30
d    40
dtype: int64
```

Using a Dictionary:

```
data = {'a': 10, 'b': 20, 'c': 30}
s = pd.Series(data)
print(s)
```

Output:

```
a    10
b    20
c    30
dtype: int64
```

---

2. `pd.DataFrame()` The `pd.DataFrame()` function is used to create a **two-dimensional labeled table** (DataFrame), similar to an Excel spreadsheet or a SQL table. A DataFrame is made up of **rows** and **columns** with both row and column labels.

Key Points:

- A **DataFrame** is essentially a collection of Series (columns) sharing the same index (rows).
- It can be created from various data structures like dictionaries, lists of lists, NumPy arrays, etc.

Example: Creating a DataFrame from a Dictionary

```
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'Los Angeles', 'Chicago']}
df = pd.DataFrame(data)
print(df)
```

Output:

```
Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

Specifying a Custom Index:

```
df = pd.DataFrame(data, index=['a', 'b', 'c'])
print(df)
```

Output:

```
Name  Age         City
a    Alice   25     New York
b      Bob   30  Los Angeles
c  Charlie   35      Chicago
```

Creating a DataFrame from a List of Lists:

```
data = [['Alice', 25, 'New York'],
        ['Bob', 30, 'Los Angeles'],
        ['Charlie', 35, 'Chicago']]
df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])
print(df)
```

Output:

```
Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

Comparison:

Function	Data Structure	Dimensionality	Use Case
`pd.Series()`	1D labeled array	1-dimensional	Best for single c of data with (like a single va
`pd.DataFrame()`	2D labeled table	2-dimensional	Best for data with row columns.

functions are fundamental building blocks for working with data in Pandas.

---

1. Data Creation

- **`pd.Series()`**: Creates a one-dimensional labeled array (Series).
- **`pd.DataFrame()`**: Creates a two-dimensional labeled table (DataFrame).

---

2. Data Input/Output (I/O)

- **`pd.read_csv()`**: Reads data from a CSV file into a DataFrame.
- **`pd.to_csv()`**: Exports a DataFrame to a CSV file.
- **`pd.read_excel()`**: Reads data from an Excel file into a DataFrame.
- **`pd.to_excel()`**: Exports a DataFrame to an Excel file.
- **`pd.read_json()`**: Reads JSON data into a DataFrame.
- **`pd.read_sql()`**: Reads data from an SQL database into a DataFrame.

---

3. Data Inspection

- `df.head(n)`: Returns the first `n` rows (default: 5).
- `df.tail(n)`: Returns the last `n` rows (default: 5).
- `df.info()`: Provides a summary of the DataFrame, including data types and non-null counts.
- `df.describe()`: Generates descriptive statistics for numeric columns.
- `df.shape`: Returns the dimensions (rows, columns) of the DataFrame.
- `df.dtypes`: Lists the data types of each column.

---

4. Data Selection

- `df['column_name']`: Accesses a single column as a Series.
- `df[['col1', 'col2']]`: Accesses multiple columns as a DataFrame.
- `df.iloc[]`: Selects rows/columns by index positions.
- `df.loc[]`: Selects rows/columns by labels or conditions.

---

5. Data Cleaning

- `df.isnull()`: Detects missing values.
- `df.notnull()`: Detects non-missing values.
- `df.fillna(value)`: Replaces missing values with the specified value.
- `df.dropna()`: Drops rows or columns with missing values.
- `df.replace(old_value, new_value)`: Replaces specific values in the DataFrame.
- `df.rename()`: Renames columns or rows.

---

6. Data Transformation

- `df.sort_values(by='column')`: Sorts rows based on the specified column.
- `df.sort_index()`: Sorts rows/columns by their index.
- `df.groupby()`: Groups data for aggregation or transformation.
- `df.pivot_table()`: Creates a pivot table for aggregating data.
- `df.apply()`: Applies a custom function to DataFrame rows or columns.
- `df.astype()`: Changes the data type of a column.

---

7. Data Aggregation

- `df.mean()`: Computes the mean of numeric columns.
- `df.sum()`: Computes the sum of numeric columns.
- `df.count()`: Counts non-NA/null entries in each column or row.
- `df.min()` / `df.max()`: Finds the minimum/maximum value.
- `df.median()`: Calculates the median of numeric columns.
- `df.mode()`: Finds the mode(s) of columns.

---

8. Merging and Reshaping

- `pd.merge()`: Merges two DataFrames based on a common column/key.
- `pd.concat()`: Concatenates two or more DataFrames along rows or columns.
- `df.join()`: Joins DataFrames on their index or a key.
- `df.melt()`: Unpivots a DataFrame from wide to long format.
- `df.pivot()`: Pivots a DataFrame from long to wide format.

---

9. Statistical Operations

- `df.corr()`: Computes correlation between numeric columns.
- `df.cov()`: Computes covariance of numeric columns.
- `df.std()`: Calculates the standard deviation.
- `df.var()`: Computes variance.
- `df.cumsum()`: Calculates the cumulative sum.

---

10. Time Series Functions

- `pd.to_datetime()`: Converts data to a datetime object.
- `df.resample()`: Aggregates time series data.
- `df.shift()`: Shifts data by a specified number of periods.

These functions make Pandas an extremely versatile tool for handling and analyzing data efficiently.

0.21.4 Panda IO

The provided Python script demonstrates how to create, manipulate, and export data using the Pandas library in various file formats. It begins by creating a sample DataFrame with columns for `Name`, `Age`, and `City`, which is then exported to a CSV file, an Excel file (using `openpyxl`), and a JSON file. The script also reads data back from these files into new DataFrames, showcasing how Pandas handles different file formats efficiently.

Additionally, the script demonstrates data manipulation by creating a copy of the original DataFrame, filtering rows where the `Age` column value is greater than 25. This process is akin to simulating a SQL-like read and write operation. The output showcases the data's consistent structure across file formats and highlights the use of Pandas for data analysis tasks. The script also mentions a potential `openpyxl` dependency issue for Excel operations and provides a solution for its installation.

```
# if not installed
# pip install pandas
# pip install openpyxl

import pandas as pd

# Create sample data to use in the program
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

# Creating a DataFrame
df = pd.DataFrame(data)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 1. Exporting to a CSV File
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
df.to_csv('data.csv', index=False)
print("Data exported to 'data.csv'")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 2. Reading from a CSV File
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
df_csv = pd.read_csv('data.csv')
print("\nData read from CSV file:")
print(df_csv)


# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 3. Exporting to an Excel File
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

df.to_excel('data.xlsx', index=False, sheet_name='Sheet1')
print("\nData exported to 'data.xlsx'")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 4. Reading from an Excel File
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')
```

```
print("\nData read from Excel file:")
print(df_excel)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 5. Exporting to a JSON File
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
df.to_json('data.json', orient='records')
print("\nData exported to 'data.json'")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 6. Reading from a JSON File
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
df_json = pd.read_json('data.json')
print("\nData read from JSON file:")
print(df_json)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 7. Copying DataFrame to Simulate SQL Read/Write
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Simulating a "write" operation by creating a new DataFrame
df_copy = df.copy()
print("\nData copied to a new DataFrame:")
print(df_copy)

# Simulating a "read" operation by modifying the copied DataFrame
df_modified = df_copy[df_copy['Age'] > 25]
print("\nFiltered DataFrame with Age > 25:")
print(df_modified)

'''
OUTPUT:
Data exported to 'data.csv'

Data read from CSV file:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago

Data exported to 'data.xlsx'

Data read from Excel file:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago

Data exported to 'data.json'

Data read from JSON file:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
```

```
2  Charlie   35      Chicago

Data copied to a new DataFrame:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago

Filtered DataFrame with Age > 25:
      Name  Age         City
1      Bob   30  Los Angeles
2  Charlie   35      Chicago

'''



'''
The error ModuleNotFoundError: No module named 'openpyxl' means that the Python module openpyx

To resolve this, follow these steps:
Install openpyxl: Run the following command in your terminal or command prompt:

pip install openpyxl
Verify Installation: After installation, verify that the module has been installed correctly b

pip show openpyxl
This should display the installed version and location of the openpyxl module.

'''
```

0.21.5 Panda Inspection

Explanation of Each Function

1. `df.head(n)`: Displays the first `n` rows of the DataFrame. If `n` is not specified, it defaults to 5.
2. `df.tail(n)`: Displays the last `n` rows of the DataFrame. If `n` is not specified, it defaults to 5.
3. `df.info()`: Provides a concise summary of the DataFrame, including the number of non-null values, data types of each column, and memory usage.
4. `df.describe()`: Generates descriptive statistics (e.g., mean, standard deviation, min, max) for all numeric columns.
5. `df.shape`: Returns the dimensions of the DataFrame as a tuple `(number of rows, number of columns)`.
6. `df.dtypes`: Displays the data type of each column in the DataFrame.

```
import pandas as pd

# Create sample data
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 22],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix'],
    'Salary': [70000, 80000, 120000, 90000, 75000]
}

# Create a DataFrame
df = pd.DataFrame(data)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Data Inspection Functions
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# 1. df.head(n): First `n` rows (default: 5)
print("First 3 rows of the DataFrame:")
print(df.head(3))
print("\n")

# 2. df.tail(n): Last `n` rows (default: 5)
print("Last 2 rows of the DataFrame:")
print(df.tail(2))
print("\n")

# 3. df.info(): Summary of the DataFrame
print("DataFrame Info:")
print(df.info())
print("\n")

# 4. df.describe(): Descriptive statistics for numeric columns
print("Descriptive Statistics:")
print(df.describe())
print("\n")

# 5. df.shape: Dimensions of the DataFrame (rows, columns)
```

```
print("Shape of the DataFrame:")
print(df.shape)
print("\n")

# 6. df.dtypes: Data types of each column
print("Data Types of Each Column:")
print(df.dtypes)
print("\n")

,,,
OUTPUT

First 3 rows of the DataFrame:
      Name  Age         City  Salary
0    Alice   25     New York   70000
1      Bob   30  Los Angeles   80000
2  Charlie   35      Chicago  120000


Last 2 rows of the DataFrame:
    Name  Age     City  Salary
3  David   40  Houston   90000
4    Eve   22  Phoenix   75000


DataFrame Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
 #   Column  Non -Null Count  Dtype
 - - -  - - - - - -  - - - - - - - - - - - - - - - - -  - - - - -
 0   Name    5 non -null      object
 1   Age     5 non -null      int64
 2   City    5 non -null      object
 3   Salary  5 non -null      int64
dtypes: int64(2), object(2)
memory usage: 292.0+ bytes
None


Descriptive Statistics:
             Age         Salary
count   5.000000       5.000000
mean   30.400000   87000.000000
std     7.300685   19874.606914
min    22.000000   70000.000000
25%    25.000000   75000.000000
50%    30.000000   80000.000000
75%    35.000000   90000.000000
max    40.000000  120000.000000


Shape of the DataFrame:
```

```
(5, 4)


Data Types of Each Column:
Name      object
Age        int64
City      object
Salary     int64
dtype: object

,,,
```

0.21.6 Panda Selection

Explanation of Each Code Block

1. `df['column_name']`:
   - Retrieves a single column as a Series.
   - Example: Access the `Name` column.
2. `df[['col1', 'col2']]`:
   - Retrieves multiple columns as a DataFrame.
   - Example: Access the `Name` and `City` columns.
3. `df.iloc[]`:
   - Selects rows and columns by integer-based index positions.
   - Example: Select the first 3 rows (index 0 to 2) and columns 1 (`Age`) and 2 (`City`).
4. `df.loc[]`:
   - Selects rows and columns by labels or applies conditions.
   - Example 1: Select rows where the `Age` column is greater than 30 and retrieve `Name` and `Salary`.
   - Example 2: Select all columns for a specific row by its index label.

```
# Data Selection
import pandas as pd

# Create sample data
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 22],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix'],
    'Salary': [70000, 80000, 120000, 90000, 75000]
}

# Create a DataFrame
df = pd.DataFrame(data)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Data Selection Examples
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# 1. Access a single column as a Series
print("Accessing the 'Name' column:")
print(df['Name'])
print("\n")

# 2. Access multiple columns as a DataFrame
print("Accessing 'Name' and 'City' columns:")
print(df[['Name', 'City']])
print("\n")

# 3. Select rows/columns by index positions using iloc
print("Selecting specific rows and columns using iloc (rows 0 to 2, columns 1 and 2):")
print(df.iloc[0:3, 1:3])  # Rows 0 to 2, columns 1 to 2
print("\n")
```

```
# 4. Select rows/columns by labels or conditions using loc
print("Selecting rows where 'Age' > 30 and specific columns using loc:")
print(df.loc[df['Age'] > 30, ['Name', 'Salary']])  # Rows with Age > 30 and specific columns
print("\n")

print("Selecting a specific row and all columns using loc (row with label 3):")
print(df.loc[3])  # Row with index label 3

'''
OUTPUT
Accessing the 'Name' column:
0      Alice
1        Bob
2    Charlie
3      David
4        Eve
Name: Name, dtype: object


Accessing 'Name' and 'City' columns:
      Name         City
0    Alice     New York
1      Bob  Los Angeles
2  Charlie      Chicago
3    David      Houston
4      Eve      Phoenix


Selecting specific rows and columns using iloc (rows 0 to 2, columns 1 and 2):
   Age         City
0   25     New York
1   30  Los Angeles
2   35      Chicago


Selecting rows where 'Age' > 30 and specific columns using
loc:
      Name  Salary
2  Charlie  120000
3    David   90000


Selecting a specific row and all columns using loc (row with label 3):
Name        David
Age            40
City      Houston
Salary      90000
Name: 3, dtype: object
'''
```

0.21.7 Panda Data Cleaning

Explanation of Each Function

1. `df.isnull()`:
   - Detects missing (NaN) values in the DataFrame.
   - Returns a DataFrame of the same shape with `True` for missing values and `False` otherwise.
2. `df.notnull()`:
   - Detects non-missing values in the DataFrame.
   - Returns a DataFrame of the same shape with `True` for non-missing values and `False` otherwise.
3. `df.fillna(value)`:
   - Replaces missing values with a specified value or dictionary of values for different columns.
4. `df.dropna()`:
   - Drops rows or columns with missing values. By default, rows with any missing values are removed.
5. `df.replace(old_value, new_value)`:
   - Replaces specific values in the DataFrame with new ones.
6. `df.rename()`:
   - Renames columns or rows. It requires a dictionary mapping old names to new names.

```
# Panda Data Cleaning

import pandas as pd
import numpy as np

# Create sample data with missing values
data = {
    'Name': ['Alice', 'Bob', 'Charlie', None, 'Eve'],
    'Age': [25, np.nan, 35, 40, 22],
    'City': ['New York', 'Los Angeles', None, 'Houston', 'Phoenix'],
    'Salary': [70000, 80000, None, 90000, 75000]
}

# Create a DataFrame
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 1. Detect missing values with isnull()
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Detecting missing values with isnull():")
print(df.isnull())
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
# 2. Detect non -missing values with notnull()
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Detecting non -missing values with notnull():")
print(df.notnull())
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 3. Fill missing values with fillna()
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Filling missing values with a default value:")
df_filled = df.fillna({'Name': 'Unknown', 'Age': 0, 'City': 'Unknown', 'Salary': 50000})
print(df_filled)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 4. Drop rows with missing values using dropna()
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Dropping rows with any missing values:")
df_dropped = df.dropna()
print(df_dropped)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 5. Replace specific values with replace()
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Replacing 'New York' with 'NYC' in the 'City' column:")
df_replaced = df.copy()
df_replaced['City'] = df_replaced['City'].replace('New York', 'NYC')
print(df_replaced)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 6. Rename columns with rename()
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Renaming columns:")
df_renamed = df.rename(columns={'Name': 'Full Name', 'Age': 'Years', 'City': 'Location'})
print(df_renamed)
print("\n")


'''
OUTPUT
Original DataFrame:
      Name   Age         City   Salary
0    Alice  25.0     New York  70000.0
1      Bob   NaN  Los Angeles  80000.0
2  Charlie  35.0         None      NaN
3     None  40.0      Houston  90000.0
4      Eve  22.0      Phoenix  75000.0


Detecting missing values with isnull():
    Name    Age   City  Salary
```

```
0  False  False  False   False
1  False   True  False   False
2  False  False   True    True
3   True  False  False   False
4  False  False  False   False


Detecting non -missing values with notnull():
    Name    Age   City  Salary
0   True   True   True    True
1   True  False   True    True
2   True   True  False   False
3  False   True   True    True
4   True   True   True    True


Filling missing values with a default value:
      Name   Age         City   Salary
0    Alice  25.0     New York  70000.0
1      Bob   0.0  Los Angeles  80000.0
2  Charlie  35.0      Unknown  50000.0
3  Unknown  40.0      Houston  90000.0
4      Eve  22.0      Phoenix  75000.0


Dropping rows with any missing values:
    Name   Age      City   Salary
0  Alice  25.0  New York  70000.0
4    Eve  22.0   Phoenix  75000.0


Replacing 'New York' with 'NYC' in the 'City'
column:
      Name   Age         City   Salary
0    Alice  25.0          NYC  70000.0
1      Bob   NaN  Los Angeles  80000.0
2  Charlie  35.0         None      NaN
3     None  40.0      Houston  90000.0
4      Eve  22.0      Phoenix  75000.0


Renaming columns:
  Full Name  Years     Location   Salary
0     Alice   25.0     New York  70000.0
1       Bob    NaN  Los Angeles  80000.0
2   Charlie   35.0         None      NaN
3      None   40.0      Houston  90000.0
4       Eve   22.0      Phoenix  75000.0
'''
```

0.21.8 Panda Data Transformation

Explanation of Each Function

1. `df.sort_values(by='column')`:
   - Sorts the DataFrame rows based on the specified column.
   - Example: Sort rows by the `Age` column.
2. `df.sort_index()`:
   - Sorts the rows or columns by their index.
   - Example: Sort rows in descending order by their index.
3. `df.groupby()`:
   - Groups the data by a specified column and applies aggregation functions (e.g., `mean`, `sum`, etc.).
   - Example: Group data by `Department` and calculate the average `Salary`.
4. `df.pivot_table()`:
   - Creates a pivot table to summarize and aggregate data.
   - Example: Show the average `Salary` grouped by `Department` and `City`.
5. `df.apply()`:
   - Applies a custom function to each element in a column or row.
   - Example: Increase each `Salary` by 10%.
6. `df.astype()`:
   - Changes the data type of a column.
   - Example: Convert the `Age` column to a string type.

```
# Data Transformation

import pandas as pd

# Create sample data
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 22],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix'],
    'Department': ['HR', 'Finance', 'HR', 'IT', 'Finance'],
    'Salary': [70000, 80000, 120000, 90000, 75000]
}

# Create a DataFrame
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 1. df.sort_values(by='column'): Sort rows by a specified column
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Sorting rows by 'Age':")
df_sorted = df.sort_values(by='Age')
```

```
print(df_sorted)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 2. df.sort_index(): Sort rows by their index
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Sorting rows by index in descending order:")
df_index_sorted = df.sort_index(ascending=False)
print(df_index_sorted)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 3. df.groupby(): Group data for aggregation
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Grouping by 'Department' and calculating mean 'Salary':")
df_grouped = df.groupby('Department')['Salary'].mean()
print(df_grouped)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 4. df.pivot_table(): Create a pivot table
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Creating a pivot table for average 'Salary' by 'Department' and 'City':")
df_pivot = df.pivot_table(values='Salary', index='Department', columns='City', aggfunc='mean')
print(df_pivot)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 5. df.apply(): Apply a custom function
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Applying a custom function to increase 'Salary' by 10%:")
df['Salary'] = df['Salary'].apply(lambda x: x * 1.1)
print(df)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 6. df.astype(): Change data type of a column
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Changing 'Age' column to string type:")
df['Age'] = df['Age'].astype(str)
print(df)
print("\n")


'''
OUTPUT
Original DataFrame:
      Name  Age  ... Department  Salary
0    Alice   25  ...         HR   70000
1      Bob   30  ...    Finance   80000
2  Charlie   35  ...         HR  120000
3    David   40  ...         IT   90000
4      Eve   22  ...    Finance   75000
```

```
[5 rows x 5 columns]


Sorting rows by 'Age':
      Name  Age  ... Department  Salary
4      Eve   22  ...    Finance   75000
0    Alice   25  ...         HR   70000
1      Bob   30  ...    Finance   80000
2  Charlie   35  ...         HR  120000
3    David   40  ...         IT   90000

[5 rows x 5 columns]


Sorting rows by index in descending order:
      Name  Age  ... Department  Salary
4      Eve   22  ...    Finance   75000
3    David   40  ...         IT   90000
2  Charlie   35  ...         HR  120000
1      Bob   30  ...    Finance   80000
0    Alice   25  ...         HR   70000

[5 rows x 5 columns]


Grouping by 'Department' and calculating mean
'Salary':
Department
Finance    77500.0
HR         95000.0
IT         90000.0
Name: Salary, dtype: float64


Creating a pivot table for average 'Salary' by 'Department' and 'City':
City         Chicago  ...  Phoenix
Department            ...
Finance          NaN  ...  75000.0
HR          120000.0  ...      NaN
IT               NaN  ...      NaN

[3 rows x 5 columns]


Applying a custom function to increase 'Salary' by 10%:
      Name  Age  ... Department    Salary
0    Alice   25  ...         HR   77000.0
1      Bob   30  ...    Finance   88000.0
2  Charlie   35  ...         HR  132000.0
3    David   40  ...         IT   99000.0
4      Eve   22  ...    Finance   82500.0
```

```
[5 rows x 5 columns]


Changing 'Age' column to string type:
      Name Age  ... Department    Salary
0    Alice  25  ...         HR   77000.0
1      Bob  30  ...    Finance   88000.0
2  Charlie  35  ...         HR  132000.0
3    David  40  ...         IT   99000.0
4      Eve  22  ...    Finance   82500.0

[5 rows x 5 columns]
'''
```

0.21.9 Panda Aggregation

Explanation of Each Function

1. `df.mean()`:
   - Calculates the average (mean) of numeric columns.
   - Ignores non-numeric columns and missing values (NaN) by default.
2. `df.sum()`:
   - Computes the sum of numeric columns.
   - Ignores non-numeric columns and missing values by default.
3. `df.count()`:
   - Counts the number of non-NA/null entries in each column.
4. `df.min()` / `df.max()`:
   - Finds the minimum/maximum value for each column. Use `numeric_only=True` to include only numeric columns.
5. `df.median()`:
   - Calculates the median of numeric columns, ignoring NaN values.
6. `df.mode()`:
   - Finds the mode(s) (most frequent value) for each column. If multiple values are modes, all are returned.

```
# Panda Data Aggregation

import pandas as pd
import numpy as np

# Create sample data
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, np.nan],
    'Salary': [70000, 80000, 120000, 90000, 75000],
    'Department': ['HR', 'Finance', 'HR', 'IT', 'Finance']
}

# Create a DataFrame
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 1. df.mean(): Compute the mean of numeric columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Mean of numeric columns:")
print(df.mean(numeric_only=True))  # Excludes non -numeric columns
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
# 2. df.sum(): Compute the sum of numeric columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Sum of numeric columns:")
print(df.sum(numeric_only=True))
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 3. df.count(): Count non -NA/null entries in each column
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Count of non -NA/null entries in each column:")
print(df.count())
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 4. df.min() / df.max(): Find the minimum and maximum values
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Minimum values of numeric columns:")
print(df.min(numeric_only=True))
print("\n")

print("Maximum values of numeric columns:")
print(df.max(numeric_only=True))
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 5. df.median(): Compute the median of numeric columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Median of numeric columns:")
print(df.median(numeric_only=True))
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 6. df.mode(): Find the mode(s) of columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Mode of each column:")
print(df.mode())
print("\n")

,,,
OUTPUT
Original DataFrame:
      Name   Age  Salary Department
0    Alice  25.0   70000         HR
1      Bob  30.0   80000    Finance
2  Charlie  35.0  120000         HR
3    David  40.0   90000         IT
4      Eve   NaN   75000    Finance


Mean of numeric columns:
Age          32.5
Salary    87000.0
dtype: float64
```

```
Sum of numeric columns:
Age          130.0
Salary    435000.0
dtype: float64


Count of non -NA/null entries in each column:
Name          5
Age           4
Salary        5
Department    5
dtype: int64


Minimum values of numeric columns:
Age          25.0
Salary    70000.0
dtype: float64


Maximum values of numeric columns:
Age           40.0
Salary    120000.0
dtype: float64


Median of numeric columns:
Age          32.5
Salary    80000.0
dtype: float64


Mode of each column:
      Name   Age  Salary Department
0    Alice  25.0   70000    Finance
1      Bob  30.0   75000         HR
2  Charlie  35.0   80000        NaN
3    David  40.0   90000        NaN
4      Eve   NaN  120000        NaN

'''
```

0.21.10 Panda Merging and Shaping

Explanation of Each Operation

1. `pd.merge()`:
   - Merges two DataFrames on a common column or key.
   - Example: `merged_df` combines `df1` and `df2` on the `ID` column using an inner join.
2. `pd.concat()`:
   - Concatenates two or more DataFrames along rows (`axis=0`) or columns (`axis=1`).
   - Example: `concat_rows` stacks `df1` and `df2` vertically, while `concat_columns` appends `df2` as additional columns to `df1`.
3. `df.join()`:
   - Joins two DataFrames based on their indices.
   - Example: `joined_df` joins `df1_indexed` and `df3_indexed` on their index values.
4. `df.melt()`:
   - Converts a DataFrame from wide format to long format by unpivoting columns.
   - Example: `melted_df` turns `Name` and `Age` columns into rows with corresponding values.
5. `df.pivot()`:
   - Converts a DataFrame from long format to wide format by pivoting on a specified column.
   - Example: `pivoted_df` pivots `melted_df` back to wide format with `Name` and `Age` as columns.

```
# Panda Merging and Shaping

import pandas as pd

# Create sample DataFrames for demonstration
df1 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35]
})

df2 = pd.DataFrame({
    'ID': [2, 3, 4],
    'City': ['New York', 'Chicago', 'Los Angeles'],
    'Salary': [80000, 120000, 75000]
})

df3 = pd.DataFrame({
    'Department': ['HR', 'Finance', 'IT'],
    'Head': ['Alice', 'David', 'Charlie']
})

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 1. pd.merge(): Merge two DataFrames on a common column
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Merging df1 and df2 on 'ID':")
merged_df = pd.merge(df1, df2, on='ID', how='inner')  # Inner join
print(merged_df)
```

```
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 2. pd.concat(): Concatenate two DataFrames along rows or columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Concatenating df1 and df2 along rows:")
concat_rows = pd.concat([df1, df2], axis=0)  # Concatenating rows
print(concat_rows)
print("\n")

print("Concatenating df1 and df2 along columns:")
concat_columns = pd.concat([df1, df2], axis=1)  # Concatenating columns
print(concat_columns)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 3. df.join(): Join two DataFrames on their index
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Joining df1 and df3 by index:")
df1_indexed = df1.set_index('Name')
df3_indexed = df3.set_index('Head')
joined_df = df1_indexed.join(df3_indexed, how='left')
print(joined_df)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 4. df.melt(): Unpivot a DataFrame from wide to long format
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Melting df1 to long format:")
melted_df = pd.melt(df1, id_vars=['ID'], value_vars=['Name', 'Age'], var_name='Attribute', val
print(melted_df)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 5. df.pivot(): Pivot a DataFrame from long to wide format
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Pivoting melted_df back to wide format:")
pivoted_df = melted_df.pivot(index='ID', columns='Attribute', values='Value')
print(pivoted_df)
print("\n")


'''
Merging df1 and df2 on 'ID':
   ID     Name  Age      City  Salary
0   2      Bob   30  New York   80000
1   3  Charlie   35   Chicago  120000


Concatenating df1 and df2 along rows:
   ID     Name   Age         City    Salary
0   1    Alice  25.0          NaN       NaN
1   2      Bob  30.0          NaN       NaN
```

```
2   3  Charlie  35.0          NaN       NaN
0   2      NaN   NaN     New York   80000.0
1   3      NaN   NaN      Chicago  120000.0
2   4      NaN   NaN  Los Angeles   75000.0


Concatenating df1 and df2 along columns:
   ID     Name  Age  ID         City  Salary
0   1    Alice   25   2     New York   80000
1   2      Bob   30   3      Chicago  120000
2   3  Charlie   35   4  Los Angeles   75000


Joining df1 and df3 by index:
         ID  Age Department
Name
Alice     1   25         HR
Bob       2   30        NaN
Charlie   3   35         IT


Melting df1 to long format:
   ID Attribute    Value
0   1      Name    Alice
1   2      Name      Bob
2   3      Name  Charlie
3   1       Age       25
4   2       Age       30
5   3       Age       35


Pivoting melted_df back to wide format:
Attribute Age     Name
ID
1          25    Alice
2          30      Bob
3          35  Charlie
'''
```

0.21.11 Panda Statistical Operations

Explanation of Each Function

1. `df.corr()`:
   - Calculates the pairwise correlation between numeric columns.
   - Correlation values range from -1 (perfect negative correlation) to 1 (perfect positive correlation).
2. `df.cov()`:
   - Computes the covariance matrix for numeric columns.
   - Covariance measures how changes in one variable are associated with changes in another.
3. `df.std()`:
   - Calculates the standard deviation for each numeric column.
   - The standard deviation measures the amount of variation or dispersion from the mean.
4. `df.var()`:
   - Computes the variance for each numeric column.
   - Variance measures the average squared deviation from the mean.
5. `df.cumsum()`:
   - Calculates the cumulative sum of numeric columns.
   - Each entry in a column is replaced with the sum of all previous entries up to that point.

```
# Panda Statistical Operations

import pandas as pd

# Create sample data
data = {
    'ID': [1, 2, 3, 4, 5],
    'Age': [25, 30, 35, 40, 45],
    'Salary': [70000, 80000, 120000, 90000, 75000],
    'Experience': [1, 3, 5, 7, 9]
}

# Create a DataFrame
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 1. df.corr(): Compute correlation between numeric columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Correlation between numeric columns:")
print(df.corr())
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 2. df.cov(): Compute covariance between numeric columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Covariance between numeric columns:")
```

```
print(df.cov())
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 3. df.std(): Calculate standard deviation of numeric columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Standard deviation of numeric columns:")
print(df.std())
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 4. df.var(): Compute variance of numeric columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Variance of numeric columns:")
print(df.var())
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 5. df.cumsum(): Calculate cumulative sum for numeric columns
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
print("Cumulative sum of numeric columns:")
print(df.cumsum())
print("\n")


,,,
OUTPUT
Original DataFrame:
   ID  Age  Salary  Experience
0   1   25   70000           1
1   2   30   80000           3
2   3   35  120000           5
3   4   40   90000           7
4   5   45   75000           9


Correlation between numeric columns:
                 ID       Age    Salary  Experience
ID          1.000000  1.000000  0.159111    1.000000
Age         1.000000  1.000000  0.159111    1.000000
Salary      0.159111  0.159111  1.000000    0.159111
Experience  1.000000  1.000000  0.159111    1.000000


Covariance between numeric columns:
                ID      Age       Salary  Experience
ID             2.5     12.5       5000.0         5.0
Age           12.5     62.5      25000.0        25.0
Salary      5000.0  25000.0  395000000.0     10000.0
Experience     5.0     25.0      10000.0        10.0


Standard deviation of numeric columns:
```

```
ID                1.581139
Age               7.905694
Salary        19874.606914
Experience        3.162278
dtype: float64


Variance of numeric columns:
ID                    2.5
Age                  62.5
Salary        395000000.0
Experience           10.0
dtype: float64


Cumulative sum of numeric columns:
   ID  Age  Salary  Experience
0   1   25   70000           1
1   3   55  150000           4
2   6   90  270000           9
3  10  130  360000          16
4  15  175  435000          25
'''
```

0.21.12 Panda Time Series

Explanation of Each Function

1. `pd.to_datetime()`:
   - Converts a column or Series to `datetime` format.
   - Example: Convert the `Date` column in the DataFrame to a `datetime` object for time series operations.
2. `df.resample()`:
   - Aggregates time series data into specified intervals (e.g., daily, weekly).
   - Example: Resample the data into 2-day intervals and sum the `Value` column within each interval.
3. `df.shift()`:
   - Shifts the data in a column forward or backward by a specified number of periods.
   - Example: Shift the `Value` column forward by one period, and backward by one period.

```
# Panda Time Series

import pandas as pd
import numpy as np

# Create sample time series data
data = {
    'Date': ['2024 -01 -01', '2024 -01 -02', '2024 -01 -03', '2024 -01 -04', '2024 -01 -05'],
    'Value': [100, 200, 300, 400, 500]
}

# Create a DataFrame
df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
print("\n")

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 1. pd.to_datetime(): Convert 'Date' column to datetime object
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
df['Date'] = pd.to_datetime(df['Date'])
print("DataFrame with 'Date' column as datetime object:")
print(df)
print("\n")

# Set the 'Date' column as the index for time series operations
df.set_index('Date', inplace=True)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 2. df.resample(): Aggregate time series data
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Resample the data to 2 -day intervals, summing the 'Value' column
resampled_df = df.resample('2D').sum()
print("Resampled DataFrame (2 -day intervals, summed values):")
print(resampled_df)
print("\n")
```

```
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# 3. df.shift(): Shift data by a specified number of periods
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Shift the 'Value' column forward by one period
df['Shifted Value'] = df['Value'].shift(1)
print("DataFrame with 'Value' column shifted forward by one period:")
print(df)
print("\n")

# Shift the 'Value' column backward by one period
df['Backward Shifted Value'] = df['Value'].shift( -1)
print("DataFrame with 'Value' column shifted backward by one period:")
print(df)
print("\n")


'''
OUTPUT
Original DataFrame:
         Date  Value
0  2024 -01 -01    100
1  2024 -01 -02    200
2  2024 -01 -03    300
3  2024 -01 -04    400
4  2024 -01 -05    500


DataFrame with 'Date' column as datetime object:
        Date  Value
0 2024 -01 -01    100
1 2024 -01 -02    200
2 2024 -01 -03    300
3 2024 -01 -04    400
4 2024 -01 -05    500


Resampled DataFrame (2 -day intervals, summed values):
            Value
Date
2024 -01 -01    300
2024 -01 -03    700
2024 -01 -05    500


DataFrame with 'Value' column shifted forward by one period:
            Value  Shifted Value
Date
2024 -01 -01    100            NaN
2024 -01 -02    200          100.0
2024 -01 -03    300          200.0
2024 -01 -04    400          300.0
2024 -01 -05    500          400.0
```

```
DataFrame with 'Value' column shifted backward by one period:
            Value  Shifted Value  Backward Shifted Value
Date
2024 -01 -01    100            NaN                   200.0
2024 -01 -02    200          100.0                   300.0
2024 -01 -03    300          200.0                   400.0
2024 -01 -04    400          300.0                   500.0
2024 -01 -05    500          400.0                     NaN
'''
```

0.21.13 Pandas and MatPlotLib

matplotlib.pyplot, often shortened to `plt`, is a submodule within the matplotlib library for Python. It provides a state-based, MATLAB-like interface for creating various data visualizations.

Think of it as a toolset that allows you to take your numerical data and turn it into informative and visually appealing charts, graphs, and plots.

Here are some key features of matplotlib.pyplot:

- **Simple and easy to use:** pyplot offers a concise and intuitive syntax, making it accessible even for beginners in data visualization.
- **Extensive plotting capabilities:** It supports a wide range of plot types, including line plots, bar charts, scatter plots, histograms, pie charts, and many more.
- **Customization options:** You can fine-tune the visual elements of your plots, including colors, fonts, axes labels, and legends, to create professional-looking visualizations.
- **Object-oriented approach:** While pyplot is state-based, it also offers object-oriented functionalities for more advanced customization and control over your plots.

Sample Data

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Dept1	10	11	12	13	14	15	16	17	18	19	20	21
Dept2	20	21	22	23	24	25	26	27	28	29	30	31
Dept3	30	31	32	33	34	35	36	37	38	39	40	41
Dept4	40	40	40	40	50	50	50	50	60	60	60	60
Dept5	50	51	52	53	54	55	56	57	58	59	60	61

Lecture Code

```
# -* - coding: utf -8 -* -
"""
Created on Fri Nov 6 11:27:09 2020

@author: jgoudy

This script demonstrates various ways to analyze and visualize data
stored in a pandas DataFrame.

"""

import numpy as np
import pandas as pd
import sys

import matplotlib.pyplot as plt


def Example1():
    """
    This function analyzes and visualizes data in a pandas DataFrame.

    """

    # Create a NumPy array with sample data.
    arr = np.array([[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21],
```

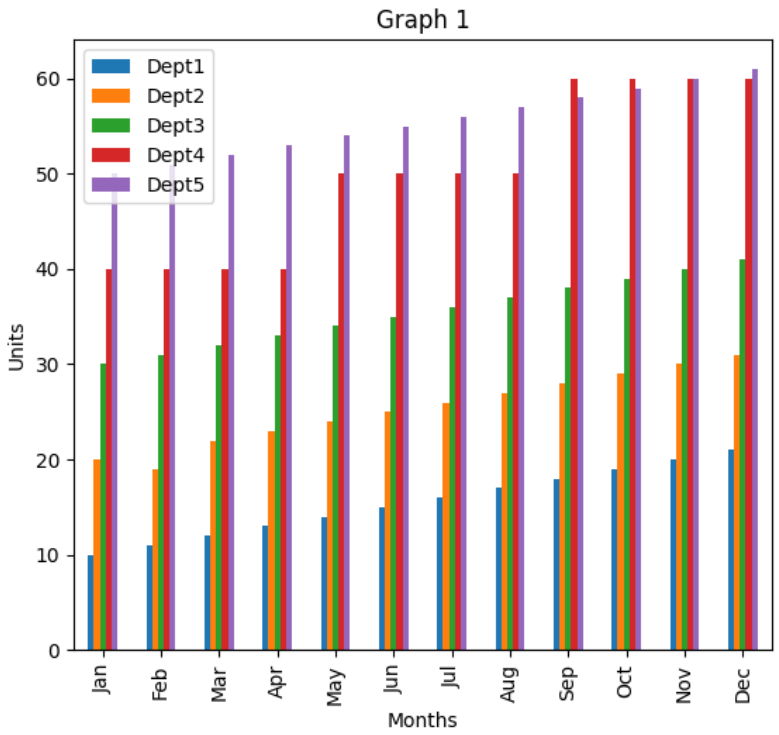
```
                [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31],
                [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41],
                [40, 40, 40, 40, 50, 50, 50, 50, 60, 60, 60, 60],
                [50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61]])

# Create a pandas DataFrame from the NumPy array.
df = pd.DataFrame(arr,
                  index=['Dept1', 'Dept2', 'Dept3', 'Dept4', 'Dept5'],
                  columns=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
                           'Aug', 'Sep', 'Oct', 'Nov', 'Dec'])

# Print descriptive statistics of the DataFrame.
print("*** df.describe ***")
print(df.describe())
print()

# Print the entire DataFrame.
print("*** df ***")
print(df)
print()

# Print the first 2 and last 2 rows of the DataFrame.
print("*** df.head(2) ***")
print(df.head(2))
print()
print("*** df.tail(2) ***")
print(df.tail(2))
print()

# Access specific columns.
print("*** df['Mar'] ***")
print(df['Mar'])
print()

# Access specific columns and rows.
print("*** df.loc[:,['Jan','Feb','Mar']] ***")
print(df.loc[:, ['Jan', 'Feb', 'Mar']])
print()

# Access specific row and columns.
print(df.loc['Dept1', ['Jan', 'Feb', 'Mar']])
print()
print((df.loc['Dept3']))
print()

# Access multiple rows.
print((df.loc[['Dept1', 'Dept3']]))
print()

# Access specific element by position.
print("*** select by position ***")
print((df.iloc[1][1]))
print()
```

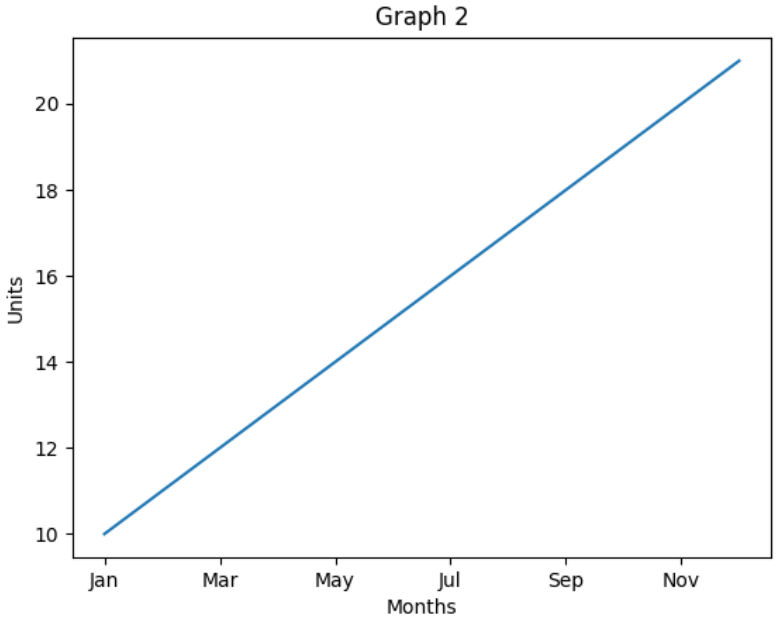
```
# Modify element by position.
df.iloc[1][1] = 19

# Get the number of rows and columns.
print(df.index.size)
print(df.columns.size)
print()

# Iterate through all elements and print them.
for r in range(df.index.size):
    for c in range(df.columns.size):
        sys.stdout.write(str(df.iloc[r][c]) + " ")
    print()

# Print the DataFrame transposed.
print("*** Transposed DataFrame ***")
print(df.T)
print()

# Verify the original DataFrame is unchanged.
print("*** Original DataFrame ***")
print(df)
print()

# Print the row and column indexes.
print("*** Row indexes ***")
print(df.index)
print()
print("*** Column indexes ***")
print(df.columns)
print()

# Start a new figure for plotting.
# plt.figure()

# Create a transposed DataFrame for easier bar graph plotting.
dft = df.T

# Plot the bar graph from the transposed DataFrame.
dft.plot(kind='bar')

# Add labels and title to the graph.
plt.xlabel('Months')
plt.ylabel('Units')
plt.title(label='Graph 1', loc='center')

# Display the first graph.
plt.show()

# Extract data for Department 1 from the transposed DataFrame.
dft1 = dft['Dept1']
```

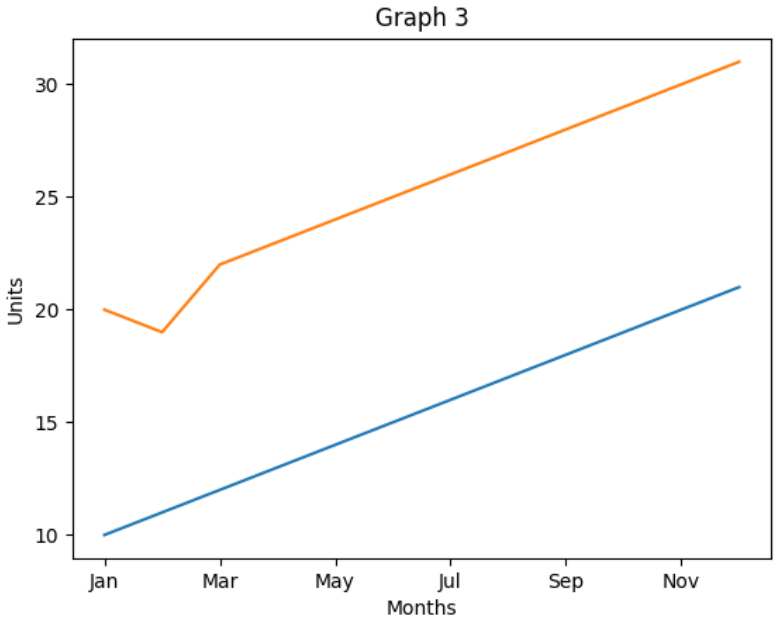
```
# Plot a bar graph for Department 1 data.
dft1.plot()

# Add labels and title to the graph.
plt.xlabel('Months')
plt.ylabel('Units')
plt.title(label='Graph 2', loc='center')

# Display the second graph.
plt.show()

# Extract data for Departments 1 and 2 from the transposed DataFrame.
dft1 = dft['Dept1']
dft2 = dft['Dept2']

# Plot a bar graph for both Department 1 and 2 data.
dft1.plot()
dft2.plot()

# Add labels and title to the graph.
plt.xlabel('Months')
plt.ylabel('Units')
plt.title(label='Graph 3', loc='center')

# Display the third graph.
plt.show()

# Note: The following graphs have similar patterns with different data combinations.

# Extract data for Departments 1 and 3.
dft1 = dft['Dept1']
dft3 = dft.Dept3

# Plot a bar graph for Department 1 with stacking and yellow color.
dft1.plot(kind='bar', stacked=True, color="yellow")

# Plot Department 2 without stacking.
dft2.plot()

# Plot Department 3 with stacking and green color.
dft3.plot(kind='bar', stacked=True, color="green")

# Add labels and title to the graph.
plt.xlabel('Months')
plt.ylabel('Units')
plt.title(label='Graph 4', loc='center')

# Display the sixth graph.
plt.show()

# Additional graphs can be created following the same pattern
# with different data combinations and plot options.
```

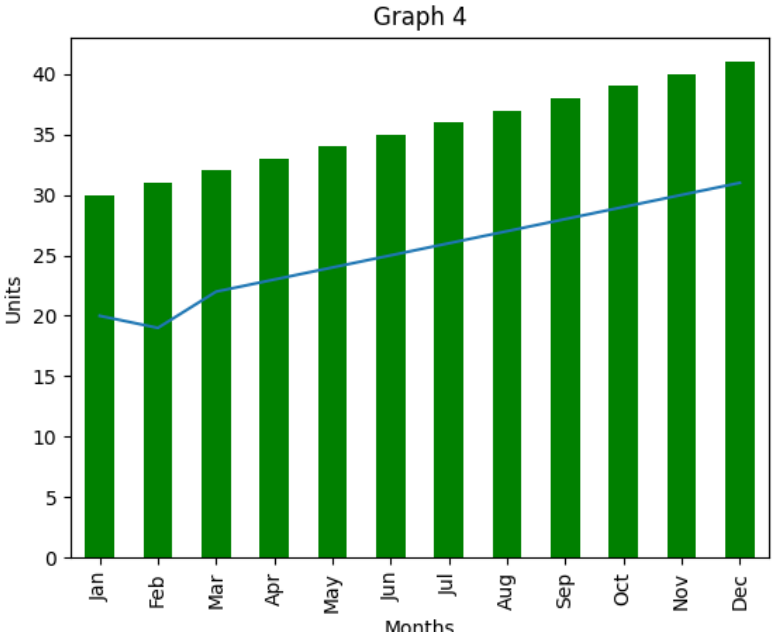
```
    # Close all open figures to avoid clutter.
    plt.close('all')

def main():

    # This function serves as the entry point for the script.
    # It calls the `Example1` function to analyze and visualize the data.


    # Clear the terminal using platform -specific methods (may not work everywhere).
    # _ = os.system('cls')
    # ipx.get_ipython().magic('clear')
    # ipx.get_ipython().magic('reset -f')

    # Run the analysis and visualization function.
    Example1()

    # end of program
    print("\nbye\n\n")

if __name__ == "__main__":
    # Only run the `main` function if the script is executed directly.
    # This prevents it from running when imported as a module.
    main()
```

Graphs




Graph 2
Units
20
18
16
14
12
10
Jan
Mar
May
Jul
Sep
Nov
Months


Graph 3
Units
30
25
20
15
10
Jan
Mar
May
Jul
Sep
Nov
Months


Graph 4
Units
0
5
10
15
20
25
30
35
40
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Months

0.21.14 Panel Data - Dog Example

Panel Data is a tool for handling mixed data types. Here is an example of some data involving dogs.

Dog Data

	Owner	Breed	Dog Name	Height	Weight	Age
Dog1	Rowan	Spaniel	Ziggy	23	19	9
Dog2	Amari	Newfoundland	Pepper	17	60	6
Dog3	Zev	Afghan	Finn	29	27	14
Dog4	Juniper	Mudi	Willow	35	43	10
Dog5	Kai	Borzoi	Scout	18	14	13
Dog6	Elodie	Terrier	Riley	32	31	2
Dog7	Silas	Terrier	Milo	12	46	5
Dog8	Nova	Leonberger	Juniper	30	22	1
Dog9	Emery	Terrier	Jasper	11	12	3
Dog10	Clementine	Spaniel	Mochi	26	16	4

Information to find

- find the dog that weighs the most (with attached data)
- find the dog that weighs the least (with attached data)
- find the average dog weight
- find only max dog weight with no other information
- list the dog names
- list dog information based on a breed

Example

Source Code

```
# examples of panel data

import sys
import pandas as pd


# global variables

# data
mydata = [
    ['Rowan', 'Spaniel', 'Ziggy', 23, 19, 9],
    ['Amari', 'Newfoundland', 'Pepper', 17, 60, 6],
    ['Zev', 'Afghan', 'Finn', 29, 27, 14],
    ['Juniper', 'Mudi', 'Willow', 35, 43, 10],
    ['Kai', 'Borzoi', 'Scout', 18, 14, 13],
    ['Elodie', 'Terrier', 'Riley', 32, 31, 2],
    ['Silas', 'Terrier', 'Milo', 12, 46, 5],
    ['Nova', 'Leonberger', 'Juniper', 30, 22, 1],
    ['Emery', 'Terrier', 'Jasper', 11, 12, 3],
    ['Clementine', 'Spaniel', 'Mochi', 26, 16, 4]
    ]

# column headers
```

```
mycols = ["Owner", "Breed", "Dog Name", "Height", "Weight", "Age"]

# indexes / row headers
myindexes = ["Dog1","Dog2","Dog3","Dog4","Dog5",
             "Dog6","Dog7","Dog8","Dog9","Dog10"]

df = None


# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def setupPanelData():

    global df

    # create a dataframe with just data
    df = pd.DataFrame(mydata)

    print("\n")
    print(df)
    print("\n")

    # add columns headers to the dataframe
    df.columns = mycols

    print("\n")
    print(df)
    print("\n")

    # add indexes / row titles to dataframe
    df.index = myindexes

    print("\n")
    print(df)
    print("\n")

    # *** Alternative Method for Above ***************************

    # create a dataframe with data, column headings
    # and row indexes at once

    df = pd.DataFrame(mydata, columns = mycols, index = myindexes)


def dogWeights():

    # This function demonstrates the various ways to get
    # a dog weight.  NOTE the ".idxmax()" will bring
    # the whole row

    # find the dog that weighs the most
    print("\nHeaviest Dog")
    maxDogWeight = df.loc[df['Weight'].idxmax()]
```

```
    print("Max Dog Weight is \n{}\n".format(maxDogWeight))
    print("{} weighs {} lbs".format(maxDogWeight['Dog Name'], maxDogWeight['Weight']))


    # find the dog that weighs the least
    print("\nLightest Dog")
    minDogWeight = df.loc[df['Weight'].idxmin()]
    print("Minimum Dog Weight is \n{}\n".format(minDogWeight))
    print("{} weighs {} lbs".format(minDogWeight['Dog Name'], minDogWeight['Weight']))

    # find the average dog weight
    print("\nAverage Dog Weight")
    meanDogWeight = df['Weight'].mean()
    print("The average dog weight is {} lbs".format(meanDogWeight))

    # find only max dog weight with no other information
    print("\nfind only max dog weight with no other information")
    maxdw= df['Weight'].max()
    print("The max dog weight is {} lbs\n".format(maxdw))


def listDogNames():

    # list the dog names
    theDogNames = df['Dog Name']
    # print the dog names
    sys.stdout.write("\nDog Names: \n")
    for c in range(len(theDogNames)):

        # print 2 names per line
        if (((c % 2) == 0) and (c > 0)):
            print()

  # print the dog name
        sys.stdout.write("{} ".format(theDogNames.iloc[c]))


    print()

def dogBreedInfo():

    # list dog information based on a breed
    # shape returns rows and columns

    # List the dog breeds
    # get the dog breeds from panel and store as list
    dgbreeds = df['Breed'].to_list()
    dgbreeds.sort()
    print()
    lastDog = None

    for i in range(len(dgbreeds)):
```

```
        # the if statement prevents duplicate
        # dogs from being printed - note
        # the list has to be sorted

        if(lastDog != dgbreeds[i] ):
            print("{}".format(dgbreeds[i]))
            lastDog = dgbreeds[i]

    # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    # flag to see whether there was any matches
    notFoundCheck = True

    # note that if the user enters nothing - "Afgan" will be the default
    dbreed = input("\nEnter the dog breed: ").lower().capitalize() or "Afghan"
    print("You entered: {}".format(dbreed))

    for r in range(df.shape[0]):
        # get row and convert it to dictionary - keys and values
        rowInfo = df.iloc[r].to_dict()

        if(rowInfo['Breed'] == dbreed):
            notFoundCheck = False

            # the key and value
            for key,value in rowInfo.items():
                sys.stdout.write("{} - {} | ".format(key,value))
            print("\n")

    # check if nothing was found
    if notFoundCheck:
        print("\nBreed was not found\n")


def menu():

    print("Please select an option:")
    print("1. Dog Weights")
    print("2. List Dog Names")
    print("3. Dog Breed Info")

    choice = input("Enter your choice (1/2/3): ")

    if choice == '1':
        dogWeights()
    elif choice == '2':
        listDogNames()
    elif choice == '3':
        dogBreedInfo()
    else:
        print("Invalid choice. Please try again.")

def main():
```

```
    setupPanelData()

    quit = "n"

    while quit != "y":

        menu()
        quit = input("Would you like to quit y/n ").lower()


    print("\nbye\n")


main()
```

Output

```
0             1        2   3   4   5
0       Rowan       Spaniel    Ziggy  23  19   9
1       Amari  Newfoundland   Pepper  17  60   6
2         Zev        Afghan     Finn  29  27  14
3     Juniper          Mudi   Willow  35  43  10
4         Kai        Borzoi    Scout  18  14  13
5      Elodie       Terrier    Riley  32  31   2
6       Silas       Terrier     Milo  12  46   5
7        Nova    Leonberger  Juniper  30  22   1
8       Emery       Terrier   Jasper  11  12   3
9  Clementine       Spaniel    Mochi  26  16   4
```

```
        Owner         Breed Dog Name  Height  Weight  Age
0       Rowan       Spaniel    Ziggy      23      19    9
1       Amari  Newfoundland   Pepper      17      60    6
2         Zev        Afghan     Finn      29      27   14
3     Juniper          Mudi   Willow      35      43   10
4         Kai        Borzoi    Scout      18      14   13
5      Elodie       Terrier    Riley      32      31    2
6       Silas       Terrier     Milo      12      46    5
7        Nova    Leonberger  Juniper      30      22    1
8       Emery       Terrier   Jasper      11      12    3
9  Clementine       Spaniel    Mochi      26      16    4
```

```
            Owner         Breed Dog Name  Height  Weight  Age
Dog1        Rowan       Spaniel    Ziggy      23      19    9
Dog2        Amari  Newfoundland   Pepper      17      60    6
Dog3          Zev        Afghan     Finn      29      27   14
Dog4      Juniper          Mudi   Willow      35      43   10
Dog5          Kai        Borzoi    Scout      18      14   13
```

```
Dog6       Elodie       Terrier    Riley      32      31    2
Dog7        Silas       Terrier     Milo      12      46    5
Dog8         Nova    Leonberger  Juniper      30      22    1
Dog9        Emery       Terrier   Jasper      11      12    3
Dog10  Clementine       Spaniel    Mochi      26      16    4


Please select an option:
1. Dog Weights
2. List Dog Names
3. Dog Breed Info
Enter your choice (1/2/3): 1

Heaviest Dog
Max Dog Weight is
Owner              Amari
Breed       Newfoundland
Dog Name          Pepper
Height                17
Weight                60
Age                    6
Name: Dog2, dtype: object

Pepper weighs 60 lbs

Lightest Dog
Minimum Dog Weight is
Owner         Emery
Breed       Terrier
Dog Name     Jasper
Height           11
Weight           12
Age               3
Name: Dog9, dtype: object

Jasper weighs 12 lbs

Average Dog Weight
The average dog weight is 29.0 lbs

find only max dog weight with no other information
The max dog weight is 60 lbs

Would you like to quit y/n n
Please select an option:
1. Dog Weights
2. List Dog Names
3. Dog Breed Info
Enter your choice (1/2/3): 2

Dog Names:
Ziggy Pepper
Finn Willow
```

```
Scout Riley
Milo Juniper
Jasper Mochi
Would you like to quit y/n n
Please select an option:
1. Dog Weights
2. List Dog Names
3. Dog Breed Info
Enter your choice (1/2/3): 3

Afghan
Borzoi
Leonberger
Mudi
Newfoundland
Spaniel
Terrier

Enter the dog breed:
You entered: Afghan
Owner - Zev | Breed - Afghan | Dog Name - Finn | Height - 29 | Weight - 27 | Age - 14 |

Would you like to quit y/n y

bye
```

0.21.15 Comparing and Contrasting NumPy Arrays and Panel Data DataFrames

Both NumPy arrays and panel data dataframes are essential tools for data manipulation and analysis in Python, but they serve distinct purposes and have different strengths and limitations. Here's a comparison:

Similarities:

- **Multidimensional data storage:** Both can store multidimensional data (vectors, matrices, higher-order arrays) efficiently.
- **Numerical operations:** Both support efficient vectorized operations like element-wise addition, multiplication, and various mathematical functions.
- **Indexing and slicing:** Both allow for flexible data access and manipulation through indexing and slicing operations.

Differences:

Data Type:

- **NumPy:** Homogeneous data type within an array (e.g., all integers, floats, etc.).
- **Panel Data DataFrames:** Heterogeneous data types across columns (e.g., numeric data, categorical data, timestamps).

Structure:

- **NumPy:** Simple n-dimensional arrays with no explicit labels or metadata.
- **Panel Data DataFrames:** Labeled axes with additional metadata like entity IDs and time periods. This structure facilitates time-series and cross-sectional analysis.

Operations:

- **NumPy:** Focuses on high-performance numerical operations and vectorized calculations.
- **Panel Data DataFrames:** Offer specialized functions for time-series analysis (e.g., lagging, differencing) and panel-specific operations (e.g., groupby, pivot_table).

Performance:

- **NumPy:** Generally faster for simple numerical operations due to its optimized C code base.
- **Panel Data DataFrames:** May be slower for basic operations due to the overhead of additional features and metadata. However, they excel in terms of data organization and efficient manipulation for panel-specific analyses.

Ease of Use:

- **NumPy:** Offers a simpler API and syntax for basic data manipulation and calculations.
- **Panel Data DataFrames:** Requires additional libraries like Pandas and Panel and may have a slightly steeper learning curve due to the richer set of features and functionalities.

Use Cases:

- **NumPy:** Ideal for scientific computing, linear algebra, and basic data analysis tasks.
- **Panel Data DataFrames:** Essential for econometrics, finance, and other fields dealing with time-series and cross-sectional data analysis.

Choosing the Right Tool:

The choice between NumPy arrays and panel data dataframes depends on your specific needs:

- For simple numerical operations and calculations on homogeneous data, NumPy offers superior performance and ease of use.
- For analyzing panel data with time-series and cross-sectional dimensions, panel data dataframes provide a structured and efficient framework with specialized functions and visualization capabilities.

Ultimately, both NumPy and panel data dataframes are valuable tools for data manipulation and analysis in Python. Understanding their strengths and limitations helps you choose the right tool for the job and unlock their full potential for insightful data exploration and analysis.

I hope this comparison clarifies the key differences between NumPy arrays and panel data dataframes. Feel free to ask if you have any further questions!

0.21.16 Pandas Quiz: Test Your Knowledge

Part 1: Data Creation

1. What function would you use to create a one-dimensional labeled array in Pandas?
   - a) `pd.DataFrame()`
   - b) `pd.Series()`
   - c) `pd.read_csv()`
   - d) `pd.to_excel()`
2. Which function creates a two-dimensional labeled table in Pandas?
   - a) `pd.DataFrame()`
   - b) `pd.Series()`
   - c) `pd.pivot()`
   - d) `pd.melt()`

Part 2: Data Input/Output (I/O)

1. Which function reads a CSV file into a DataFrame?
   - a) `pd.read_csv()`
   - b) `pd.read_sql()`
   - c) `pd.read_excel()`
   - d) `pd.to_csv()`
2. If you need to export a DataFrame to an Excel file, which function should you use?
   - a) `pd.read_excel()`
   - b) `pd.to_csv()`
   - c) `pd.to_excel()`
   - d) `pd.read_csv()`

Part 3: Data Inspection

1. Which function provides a summary of a DataFrame, including data types and non-null counts?
   - a) `df.head()`
   - b) `df.describe()`
   - c) `df.info()`
   - d) `df.shape`
2. How would you return the dimensions (number of rows and columns) of a DataFrame?
   - a) `df.describe()`
   - b) `df.shape`
   - c) `df.dtypes`
   - d) `df.head()`

Part 4: Data Selection

1. What function allows you to select rows or columns based on index positions?

   - a) `df.loc[]`
   - b) `df.iloc[]`
   - c) `df['column_name']`
   - d) `df.groupby()`

2. How would you access multiple columns (e.g., 'col1' and 'col2') in a DataFrame?

   - a) `df[['col1', 'col2']]`
   - b) `df['col1']`
   - c) `df.loc[]`
   - d) `df.iloc[]`

Part 5: Data Cleaning

1. Which function detects missing values in a DataFrame?

   - a) `df.dropna()`
   - b) `df.notnull()`
   - c) `df.isnull()`
   - d) `df.fillna()`

2. What function replaces missing values with a specified value?

   - a) `df.dropna()`
   - b) `df.fillna(value)`
   - c) `df.rename()`
   - d) `df.replace()`

Part 6: Data Transformation

1. To sort rows of a DataFrame based on a specific column, which function would you use?

   - a) `df.groupby()`
   - b) `df.sort_index()`
   - c) `df.sort_values(by='column')`
   - d) `df.apply()`

2. Which function groups data for aggregation or transformation?

   - a) `df.pivot_table()`
   - b) `df.groupby()`
   - c) `df.astype()`
   - d) `df.corr()`

Part 7: Data Aggregation

1. Which function computes the mean of numeric columns in a DataFrame?
   - a) `df.sum()`
   - b) `df.mean()`
   - c) `df.mode()`
   - d) `df.median()`
2. How would you calculate the cumulative sum of numeric values in a DataFrame?
   - a) `df.cumsum()`
   - b) `df.sum()`
   - c) `df.mean()`
   - d) `df.count()`

Part 8: Merging and Reshaping

1. If you need to combine two DataFrames based on a common column, which function should you use?
   - a) `pd.concat()`
   - b) `pd.merge()`
   - c) `df.join()`
   - d) `df.melt()`
2. What function converts a DataFrame from wide format to long format?
   - a) `pd.pivot()`
   - b) `df.melt()`
   - c) `df.pivot_table()`
   - d) `pd.concat()`

Part 9: Statistical Operations

1. Which function calculates the correlation between numeric columns?
   - a) `df.cov()`
   - b) `df.var()`
   - c) `df.corr()`
   - d) `df.std()`
2. How would you compute the variance of numeric columns?
   - a) `df.corr()`
   - b) `df.var()`
   - c) `df.cumsum()`
   - d) `df.median()`

Part 10: Time Series Functions

1. What function converts data into a datetime object in Pandas?

   - a) `pd.to_datetime()`
   - b) `df.resample()`
   - c) `df.shift()`
   - d) `df.groupby()`

2. Which function aggregates time series data in Pandas?

   - a) `df.shift()`
   - b) `pd.to_datetime()`
   - c) `df.resample()`
   - d) `df.groupby()`

Answer Key (for self-assessment):

1. b | 2. a | 3. a | 4. c | 5. c | 6. b | 7. b | 8. a | 9. c | 10. b | 11. c | 12. b | 13. b | 14. a | 15. b | 16. b | 17. c | 18. b | 19. a | 20. c

0.22 Input / Output - Files And Folders

File input/output (I/O) is the process of reading and writing data to files. This is a fundamental operation in computer science, as it allows programs to store data persistently and retrieve it later.

There are two main types of file I/O:

- **Text file I/O:** This type of I/O involves reading and writing text-based data. Text files are typically stored in plain text format, which means that they can be easily read and edited by humans.
- **Binary file I/O:** This type of I/O involves reading and writing binary data. Binary files are typically used to store non-text data, such as images, audio, and video.

File I/O is typically performed using the following steps:

1. **Open the file:** This step creates a connection to the file and prepares it for reading or writing.
2. **Read or write data:** This step reads data from the file or writes data to the file.
3. **Close the file:** This step releases the connection to the file and closes it.

Here is an example of how to read a text file using the Python programming language:

```
with open('file.txt', 'r') as file:
  data = file.read()
  print(data)
```

This code will read the contents of the file `file.txt` and print it to the console.

Here is an example of how to write a text file using the Python programming language:

```
with open('file.txt', 'w') as file:
  file.write('Hello, world!')
```

This code will write the string `Hello, world!` to the file `file.txt`.

File I/O is an important concept in computer science, as it allows programs to store data persistently and retrieve it later. File I/O is used in a wide variety of applications, including:

- **Saving and loading data:** Programs can use file I/O to save data to files and load it later. This is useful for storing data that needs to be persisted across program runs.
- **Communicating with other programs:** Programs can use file I/O to communicate with each other by exchanging data through files. This is useful for tasks such as sharing data or exchanging configuration settings.
- **Creating and modifying files:** Programs can use file I/O to create and modify files. This is useful for tasks such as creating log files or generating reports.

File I/O is a fundamental operation in computer science, and it is essential for any program that needs to store data persistently or communicate with other programs or systems.

0.22.1 Lecture Code

```
# -* - coding: utf -8 -* -
"""
Created on Wed Mar 30 10:47:56 2022

@author: jgoudy

James Goudy
"""
import os
import subprocess
```

```
import shutil


# All uservariables begin with x

# globals
xDirName = ""
xBasePath = ""
xFilePath = ""

def makeDirectory():

    # tell the function to use the global variables

    global xDirName
    global xBasePath
    global xFilePath


    # having a basename is helpful if
    # you have to make multiple folders
    # in a particualar folder

    # join the base with the directory name

    xpath = os.path.join(xBasePath,xDirName)
    print("xpath: "+ xpath)

    # make the directory

    os.mkdir(xpath)

    # Example of making directories
    # xpath value is currently c:/z/zmyDir
    # The following code will create a folder
    # for each letter in the alphabet

    xBasePath = xpath + "/"

    xletters = "abcdefghijklmnopqrstuvwxyz"

    for xx in xletters:
        os.mkdir((xBasePath + xx))

    print(xBasePath)

    input("1. Check if letter folders created.\nPress enter to continue")

    # note will will add projects
    # to c:/z/zmyDir/misc/projects
    # note we did not create a misc directory
    # using makedirs will allow us to
    # create the missing folder automatically
```

```
    os.makedirs("c:/z/zmyDir/misc/projects")

    input("2. Check for the misc folder\nPress enter to continue")

    # check if directory exists

    xstatus = os.path.isdir("c:/z/zmyDir/misc/projects")
    if(xstatus):
        print("\nDirectory exists\n")
    else:
        print("\nDirectory does not exists")


    xstatus = os.path.isdir("c:/z/zmyDir/misc/bubba")
    if(xstatus):
        print("\nDirectory exists\n")
    else:
        print("\nDirectory does not exists")



def addFile():

    # tell the function to use the global variables


    global xDirName
    global xBasePath
    global xFilePath

    print(xBasePath)

    # create file path


    xfilename = "xfile1.txt"
    xpath = xBasePath+ "a/" + xfilename

    # xpath = xpath + xfilename

    print(xpath)

    # add a file to the "a" folder

    # 'w' create a file - creates new file if it doesn't exist
    # overwrites data if it does exist

    xfw = open(xpath, 'w')


    """
    Read Only ('r'): Open text file for reading. If the file does not exist,
```

```
raises I/O error. This is also the default mode in which
the file is opened.

Read and Write ('r+'): Open the file for reading and writing.
Raises I/O error if the file does not exist.

Write Only ('w'): Open the file for writing.
Existing data is truncated and over -written.
Creates the file if the file does not exist.

Write and Read ('w+'): Open the file for reading and writing.
Existing data is truncated and over -written.

Append Only ('a'): Open the file for writing.
The file is created if it does not exist.
Data is added to the end of the file.

Append and Read ('a+'): Open the file for reading and writing.
The file is created if it does not exist.
Data is added to the end of the file.
"""

input("3. Check to see if file was created\nPress enter to continue")

# write text to file
for c in range(10):
    xfw.write(str(c+1) + ". Sample text\r\n")

# close the file when done writing
xfw.close()

# append to the file
xfw = open(xpath, 'a')

# add some text to the end
for c in range(10):
    xfw.write(str(c+1) + "z. ZZZ Sample text\r\n")

# close the file when done writing
xfw.close()

xFilePath = xpath
print("xxx: " + xpath)

# In order to use subprocess.Popen which opens the file directory
# the slashes have to be reversed
xpath = swapSlash(xpath)

# subprocess will open the operating system file explorer in windows
# c:/z/zmyDir/a/
# subprocess.Popen(r'explorer /select,"C:\z\zmyDir\a\xfile1.txt"')
subprocess.Popen(r'explorer /select,"' + xpath + '"' )
```

```
    input("4. Exploer window opened\nPress enter to continue")


def deleteFile():

    # delete a file

    # tell the function to use the global variables

    global xFilePath

    print("Delete / remove : " + xFilePath)

    # check if file exists and remove it
    if os.path.exists(xFilePath):
        os.remove(xFilePath)
    else:
        print("File could not be found and was not deleted")

    # you can also use the os.unlink command as well to delete a file

    input("5. Check if file was deleted\nPress enter to continue")

def deleteFolders():

    '''
    remove() throws an exception if the file doesn't exist,
    while shutil. rmtree() doesn't care if the directory is empty or not.
    '''

    # remove the 'a' directory
    # os.chmod('c:\\z\\zmyDir\\b', stat.S_IXOTH)
    os.rmdir('c:\\z\\zmyDir\\b')

    input("5a. Check if b folder was deleted\n" + \
          "Press enter to continue")

    # remove all directories
    choice = input("Delete all directories y/n : ").lower()

    if choice == "y":
        try:
            xtemp = "c:\\z\\zmyDir\\"
            shutil.rmtree(xtemp)
            print("ok")

        except Exception as e:
           print(e)

    input("5. Check if all created folders were deleted\n" + \
          "Press enter to continue")

def swapSlash(aString):
```

```
    newString = ""

    for l in aString:
        if l == "/":
            newString = newString + "\\"
        else:
            newString = newString + l

    # print(newString)

    return newString


def main():

    print("working\n")

    # access global variables

    global xDirName
    global xBasePath


    # directory name variable

    xDirName = "zmyDir"

    # basename
    # in this case make sure you have a
    # z folder on your c drive

    xBasePath = "c:/z/"

    # remove the example code if previous ran and still existing
    try:
        xtemp = xBasePath+xDirName
        shutil.rmtree(xtemp)

    except Exception as e:
       print(e)


    # main program code

    makeDirectory()

    addFile()

    deleteFile()

    deleteFolders()
```

```
    print("\nbye bye")

main()
```

0.23 Tkinter

Tkinter stands for *Tk interface*. **tkinter** is the standard Python interface for creating Graphical User Interfaces (GUI). It can run on Windows, macOS, and Linux.

> **Note**
> The **Geometry** is the command to actually place the form on the screeen.

0.23.1 Lecture Code

```
"""
calculator

NOTE - MOST OF THE FORM CONTROLS/WIDGETS ARE
WRITTEN AS GLOBAL

"""

import tkinter as tk
from tkinter import messagebox
import sys


def addBoxes():

    try:
        ans =  float(inNum1.get("1.0", "end")) + \
             float(inNum2.get("1.0", tk.END))
    except:
        ans = "Error"

    lblAns["text"] = ans


def addButtonClick(event):
    addBoxes()


def focus_next_widget(event):
    event.widget.tk_focusNext().focus()
    return("break")


def boxClear(event):
    clearBoxes()


def clearBoxes():

    # the clearBoxes is also used in a "non"
    # event function

    inNum1.delete("1.0", "end")
```

```
    inNum2.delete("1.0", "end")
    lblAns["text"] = ""


def close_window():

    # if you don't want to show the messagebox
    # the if statement can be deleted. The destroy
    # and exit commands are still required.
    # Tab them appropriately.
    if messagebox.askokcancel("Quit", "Do you want to quit?"):

        # these two are always required
        theFrame.destroy()
        sys.exit(0)

# sometimes you will it root
theFrame = tk.Tk()


# add title
theFrame.title("Calculator")

# set the height and width
frameWidth = 250
frameHeight = 150

# get the screen width and height
scrnH = theFrame.winfo_screenheight()
scrnW = theFrame.winfo_screenwidth()

# calculate the center of window
xpos = int((scrnW/2) - (frameWidth/2))
ypos = int((scrnH/2) - (frameHeight/2))

# set geometry - This command places the frame on the screen with the
# hegith x width and the position xpos, ypos in the window
# theFrame.geometry('250x150+200+200')

# Note the code is building a string
# 250X150 + xpos + ypos
theFrame.geometry(str(frameWidth)+"x" + str(frameHeight)+"+" +
                  str(xpos)+"+"+str(ypos))

# frame widgets need to be global

# label 1
tk.Label(theFrame, text="Num 1", bg="#3300CC", fg="white").place(x=10, y=10)

# add an input box 1
inNum1 = tk.Text(theFrame, height=1, width=10)
inNum1.place(x=100, y=10)
inNum1.bind("<Tab>", focus_next_widget)
```

```
# label 2
tk.Label(theFrame, text="Num 2", bg="#3300CC", fg="white").place(x=10, y=40)

# add an input box 1
inNum2 = tk.Text(theFrame, height=1, width=10)
inNum2.place(x=100, y=40)
inNum2.bind("<Tab>", focus_next_widget)

bttnAdd = tk.Button(theFrame, text="Add", height=1, width=10, command=addBoxes)
bttnAdd.place(x=100, y=70)

# bind button to multiple events
bttnAdd.bind("<Return>", addButtonClick)
bttnAdd.bind("<Button -1>", addButtonClick)
bttnAdd.bind("<Tab>", focus_next_widget)

# add answer label
lblAns = tk.Label(theFrame, text="Answer")
lblAns.place(x=10, y=70)

# add clear button
bttnClr = tk.Button(theFrame, text="Clear", fg="#000080",
                    height=1, width=10, command=clearBoxes)
bttnClr.place(x=100, y=100)

# bind our bind to events
bttnClr.bind("<Return>", boxClear)
bttnClr.bind("<Button -1>", boxClear)
bttnClr.bind("<Tab>", focus_next_widget)

# use this to close the window
# note that you have to write the close_window function
theFrame.protocol("WM_DELETE_WINDOW", close_window)

# main loop
theFrame.mainloop()
```

0.24 Tic Tac Toe

0.24.1 Lecture Code

```
"""
Programmer: James Goudy

Homage To The Panda

Display the Board
userTurn
computerTurn
Check for win or end

User is X
User goes first

"""
import sys
import random

# global variables
Board = ["1", "2", "3", "4", "5", "6", "7", "8", "9"]
GameMoves = 0
GameLoop = True


def displayBoard():
    print("Game Board")

    for c in range(9):
        if((c > 0) and (c % 3 == 0)):
            print()
        sys.stdout.write(Board[c].ljust(3))

    print("\n")


def resetVariables():
    global Board
    global GameMoves
    global GameLoop

    Board = ["1", "2", "3", "4", "5", "6", "7", "8", "9"]
    GameMoves = 0
    GameLoop = True

def userTurn():

    global GameMoves

    try:
        print("\nUser's Turn")
```

```
        displayBoard()

        userChoice = int(input("Please choose a square: ")) - 1

        if(Board[userChoice] != "X" and Board[userChoice] != "O"):
            Board[userChoice] = "X"
        else:
            # prevent cheating
            userTurn()

        displayBoard()

        GameMoves += 1

    except:
        print("Bad Choice - Please Try Again")
        userTurn()

def computerTurn():

    run = True
    global GameMoves

    while(run):
        ct = random.randint(0, 8)

        if(Board[ct] != "X" and Board[ct] != "O"):
            Board[ct] = "O"
            run = False
            GameMoves += 1

    displayBoard()


def checkForWinOrEnd(playerXO):

    global GamesMoves
    global GameLoop

    if(Board[0] == Board[1] == Board[2] == playerXO):
        print(playerXO + " wins")
        GameLoop = False
        return
    elif(Board[3] == Board[4] == Board[5] == playerXO):
        print(playerXO + " wins")
        GameLoop = False
        return
    elif(Board[6] == Board[7] == Board[8] == playerXO):
        print(playerXO + " wins")
        GameLoop = False
        return
    elif(Board[0] == Board[3] == Board[6] == playerXO):
        print(playerXO + " wins")
```

```
        GameLoop = False
        return
    elif(Board[1] == Board[4] == Board[7] == playerXO):
        print(playerXO + " wins")
        GameLoop = False
        return
    elif(Board[2] == Board[5] == Board[8] == playerXO):
        print(playerXO + " wins")
        GameLoop = False
        return
    elif(Board[0] == Board[4] == Board[8] == playerXO):
        print(playerXO + " wins")
        GameLoop = False
        return
    elif(Board[2] == Board[4] == Board[6] == playerXO):
        print(playerXO + " wins")
        GameLoop = False
        return

    if GameMoves == 9:
        GameLoop = False
        print("Tie - no winner")


def main():

    choice = "n"

    while choice != "y":

        displayBoard()

        while(GameLoop):
            userTurn()
            checkForWinOrEnd("X")

            if GameLoop == False:
                break

            computerTurn()
            checkForWinOrEnd("O")

        choice = input("Would you like quit? y/n ").lower()
        resetVariables()

main()
```

0.25 Serialization

0.25.1 Definition

Serialization is the process of transforming data structures or objects into a format that can be:

- Stored (e.g., files in databases or storage devices)
- Transmitted (e.g., data streams over networks)

This format allows for reconstruction of the data later, potentially even in a different computer environment. The resulting data, after serialization, is typically a sequence of bytes.

There's a companion process called **deserialization**, which is essentially the opposite. It takes the serialized byte stream and recreates the original data structure or object. Serialization and deserialization work together to make data portable and usable across different systems.

0.25.2 Serialization Types

There are several major forms of serialization used in Python, each with its own advantages and purposes:

1. **Pickle:** This is the built-in standard library for serialization in Python. It's very powerful and can handle most native data types, including custom classes, objects and functions. The serialized data is in a binary format, making it compact but not human-readable. **Use pickle when:**
    - You need to serialize complex Python objects for internal use within your application.
    - Speed and efficiency are a priority.
2. **JSON (JavaScript Object Notation):** This is a popular text-based format based on key-value pairs. It's human-readable and language-agnostic, making it a good choice for data exchange between different programming languages and systems. JSON can handle most basic data types like dictionaries, lists, strings, and numbers. **Use JSON when:**
    - You need to exchange data with other applications or APIs.
    - Human-readability of the serialized data is important.
3. **XML (Extensible Markup Language):** Another text-based format with a hierarchical structure using tags. It's more verbose than JSON but offers more flexibility for complex data structures. XML is widely used for data interchange and configuration files. **Use XML when:**
    - You need to exchange data with systems that specifically require XML format.
    - You need a more structured format for complex data hierarchies.
4. **YAML (YAML Ain't Markup Language):** A human-readable data serialization format similar to JSON but with a simpler syntax. It's a good compromise between readability and compactness. **Use YAML when:**
    - You want a more concise text-based format compared to XML.
    - You prioritize human-readability while maintaining some structure for configuration files.
5. **CSV (Comma-Separated Values):** A simple text format where data is stored in rows and columns, separated by commas. It's lightweight and easy to parse, but limited to basic data types. **Use CSV when:**
    - You need a very simple format for exchanging tabular data.
    - Compatibility with spreadsheet software is important.
6. **HDF5 (Hierarchical Data Format):** This format is specifically designed for storing large datasets, especially scientific data with complex structures. It offers efficient storage for multi-dimensional arrays, large matrices, and other scientific data types. HDF5 files can also store metadata along with the data itself. **Use HDF5 when:**

- You're working with large scientific datasets with complex structures.
- Efficient storage and retrieval of multi-dimensional data is crucial.
- You need to store metadata alongside your data.
-

The choice of serialization format depends on your specific needs. Consider factors like:

- **Data types:** Can the format handle the data structures you need to serialize?
- **Readability:** Do you need the serialized data to be human-readable?
- **Interoperability:** Will the data be exchanged with other systems?
- **Performance:** How important is serialization speed and efficiency?

Format Types

Here's a table summarizing the key points:

Format	Description	Advantages	Disadvantages	Use Cases
Pickle	Built-in Python library	Powerful, handles complex objects	Binary, not human-readable	Internal data exchange
JSON	Text-based, key-value pairs	Human-readable, language-agnostic	Limited data types	Data exchange between applications/APIs
XML	Text-based, hierarchical structure	Flexible for complex data	Verbose compared to JSON	Data interchange, configuration files
YAML	Human-readable data format	Concise syntax compared to XML	Less common than JSON/XML	Configuration files
CSV	Simple text format, comma-separated values	Lightweight, easy to parse	Limited to basic data types	Tabular data exchange, spreadsheet compatibility
HDF5	Designed for large scientific datasets	Efficient storage, multi-dimensional arrays, metadata Can also be used across different languages including C++ and Java	More complex setup compared to simpler formats	Scientific computing, large data analysis

0.25.3 Serialization - CSV

CSV Files and the Python csv Module

What is a CSV File? A CSV (Comma-Separated Values) file is a simple text-based format for storing tabular data. Each line in a CSV file represents a record, and each record consists of one or more fields separated by commas. This format is widely used for data exchange between different applications and systems because of its simplicity and readability.

Example CSV file:

```
Name,Title,Salary,City
Ally,CEO,2000000,Kalispell
Bob,CFO,1000000,Polson
Charlie,Developer,300000,Big Fork
Daisy,Sec Tech,400000,Libby
```

Python's csv Module https://docs.python.org/3/library/csv.html#

Python's csv module provides a convenient way to work with CSV files. It offers classes for reading and writing CSV data, making it easy to parse and manipulate data in this format.

Key functionalities of the csv module:

- Reading CSV files:
  - csv.reader: Reads CSV data as a list of lists, where each inner list represents a row.
  - csv.DictReader: Reads CSV data as a list of dictionaries, where each dictionary represents a row and keys are the column headers.
- Writing CSV files:
  - csv.writer: Writes data to a CSV file as a list of lists.
  - csv.DictWriter: Writes data to a CSV file as a list of dictionaries.

Common use cases for the csv module:

- **Data import/export:** Loading data from CSV files into Python programs, and saving data from Python programs to CSV files.
- **Data cleaning and transformation:** Processing CSV data to clean, filter, or transform it before further analysis.
- **Data analysis:** Using CSV data as input for data analysis and visualization tasks.
- **Data integration:** Combining data from multiple CSV files into a single dataset.

Additional features:

- **Dialect customization:** The csv module allows you to define custom dialects for CSV files with different delimiters or quoting conventions.
- **Error handling:** The csv module provides mechanisms for handling errors that may occur during CSV parsing or writing.

Lecture Code

```
"""
Programmer: James Goudy
Program: CSV Demo

Documentation: https://docs.python.org/3/library/csv.html#

"""

import csv
import sys
```

```
def write_csv(filename, data):
  """Writes data to a CSV file.

  Args:
    filename: The name of the CSV file to write to.
    data: A list of lists, where each inner list represents a row of data.
  """

  # remember that file open attribute of 'a' will
  # allow the file to be appended
  with open(filename, 'w', newline='') as csvfile:
    csv_writer = csv.writer(csvfile)
    csv_writer.writerows(data)
    csvfile.close()

def read_csv(filename):

  """Reads data from a CSV file.

  Args:
    filename: The name of the CSV file to read from.

  Returns:
    A list of lists, where each inner list represents a row of data.
  """

  with open(filename, 'r') as csvfile:
    csv_reader = csv.reader(csvfile)
    data = list(csv_reader)
    csvfile.close()
  return data

def write_Dict(data, filename,thefieldnames) :
  with open(filename,'w',newline='')as csvfile:
    writer = csv.DictWriter(csvfile,fieldnames=thefieldnames)
    writer.writeheader()
    writer.writerows(data)

def read_dict(filename,thefieldnames):
  with open(filename,'r') as csvfile:
    reader = csv.DictReader(csvfile,thefieldnames)
    data = list(reader)
    return data

def display_dict(dataDict):

  print("\n - - - Data Dictionary Values - - -")
  tempDict = dict()

  for arow in dataDict:

    arow = dict(arow)
```

```
    for key, val in arow.items():
      sys.stdout.write(str(val).ljust(12,' ') + "\t")
    print()

def main():
    # Example usage:
    data = [
        ['Name', 'Title', 'Salary','City'],
        ['Ally', 'CEO', 2000000, 'Kalispell'],
        ['Bob', 'CFO', 1000000,'Polson'],
        ['Charlie', 'Developer', 300000, 'Big Fork'],
        ['Daisy', 'Sec Tech',400000, 'Libby']
    ]

    data2_fieldNames = ['Name', 'Title', 'Salary','City']
    data2 = [
        {'Name':'Ally2', 'Title':'CEO', 'Salary':2000000, 'City':'Kalispell'},
        {'Name':'Bob2', 'Title':'CFO', 'Salary':1000000,'City':'Polson'},
        {'Name':'Charlie2', 'Title':'Developer', 'Salary':300000, 'City':'Big Fork'},
        {'Name':'Daisy2', 'Title':'Sec Tech','Salary':400000, 'City':'Libby'}
    ]

    # filenames
    filename1 = "csv_example.txt"
    filename2 = "csv_example_dict.txt"

    # reading and writing csv file
    write_csv(filename1, data)
    read_data = read_csv(filename1)

    print("\n - - - Dictionary Variable - read_data - - -")
    print(read_data)

    print("\n - - - Iterate Through Data - - -")
    # iterate through each row
    for arow in read_data:
      # iterate through each item in that row
      for aitem in arow:
        sys.stdout.write(aitem.ljust(12,' ') + "\t")
      sys.stdout.write("\n")

    print("\n - - - Dictionary - - -")

    write_Dict(data2,"example_dict.csv",data2_fieldNames)
    thedata = read_dict("example_dict.csv",data2_fieldNames)

    print("\n - - - Dictionary Variable - theData - - -")
    print(thedata)

    print("\n - - - iterate through data dictionary variable - - -")
    display_dict(thedata)

    print("\n\nBye CSV\n")
```

```
main()

'''
 - - - Dictionary Variable - read_data - - -
[['Name', 'Title', 'Salary', 'City'], ['Ally', 'CEO', '2000000', 'Kalispell'], ['Bob', 'CFO',
Fork'], ['Daisy', 'Sec Tech', '400000', 'Libby']]

 - - - Iterate Through Data - - -
Name            Title           Salary          City
Ally            CEO             2000000         Kalispell
Bob             CFO             1000000         Polson
Charlie         Developer       300000          Big Fork
Daisy           Sec Tech        400000          Libby

 - - - Dictionary - - -

 - - - Dictionary Variable - theData - - -
[{'Name': 'Name', 'Title': 'Title', 'Salary': 'Salary', 'City': 'City'}, {'Name': 'Ally2', 'Ti

 - - - iterate through data dictionary variable - - -

 - - - Data Dictionary Values - - -
Name            Title           Salary          City
Ally2           CEO             2000000         Kalispell
Bob2            CFO             1000000         Polson
Charlie2        Developer       300000          Big Fork
Daisy2          Sec Tech        400000          Libby


Bye CSV
'''
```

Program Summary

The program is a CSV (Comma-Separated Values) demonstration script written by James Goudy. It shows how to work with CSV files for storing and manipulating tabular data. Here's a breakdown of its functionalities:

Functions:

- **write_csv(filename, data):** Writes a list of lists (where each inner list represents a row) to a CSV file.
- **read_csv(filename):** Reads data from a CSV file and returns it as a list of lists.
- **write_Dict(data, filename, thefieldnames):** Writes a list of dictionaries to a CSV file, where each dictionary represents a row and keys define column names.
- **read_dict(filename, thefieldnames):** Reads data from a CSV file containing dictionaries and returns it as a list of dictionaries.
- **display_dict(dataDict):** Iterates through a list of dictionaries and prints their values in a formatted table.

Main Program:

1. Defines sample data as lists and dictionaries with employee information (Name, Title, Salary, City).
2. Demonstrates writing and reading a regular CSV file (write_csv & read_csv functions).

3. Shows iterating through the read data and printing it in a table format.
4. Demonstrates writing and reading a CSV file with dictionaries (write_Dict & read_dict functions).
5. Uses the display_dict function to print the data from the dictionary in a formatted table.

0.25.4 Serialization - JSON

JSON serialization is the process of converting complex data structures, like objects or arrays, into a simple text format called JSON (JavaScript Object Notation). This text-based representation is easy to store, transmit, and parse.

Why Use JSON Serialization?

- **Data Exchange:** It's widely used for data interchange between different systems and applications.
- **Data Storage:** JSON files can be used to store data persistently.
- **Data Transmission:** It's efficient for sending data over networks.
- **Readability:** JSON is human-readable, making it easy to inspect and debug.

How Does It Work?

1. **Data Structure:** You have a complex data structure, like an object with properties and values.
2. **Serialization:** A serializer converts this data structure into a JSON string. This string adheres to specific JSON syntax rules.
3. **Transmission or Storage:** The JSON string can be sent over a network or stored in a file.
4. **Deserialization:** When you need to use the data again, a deserializer converts the JSON string back into its original data structure.

Key Points

- JSON is a lightweight data-interchange format.
- Serialization converts data to JSON, deserialization converts JSON back to data.
- Widely used in web applications, APIs, and data storage.

Lecture Code

```
import json

# Define a class to represent a person with name, city, state
class Person:
  def __init__(self, name, city, state):
    self.name   = name
    self.city   = city
    self.state  = state

# Function to serialize a list of people to JSON
def serialize_to_json(people):
  data = []
  for person in people:
    data.append({"name": person.name, "city": person.city,"state":person.state})
  with open("people.json", "w") as f:
    json.dump(data, f)

# Function to deserialize JSON data back to a list of people
def deserialize_from_json():
  people = []
  try:
    with open("people.json", "r") as f:
      data = json.load(f)
```

```
    for item in data:
      people.append(Person(item["name"], item["city"],item["state"]))
  except FileNotFoundError:
    print("Error: 'people.json' file not found.")
  return people

def main():
  # Example usage
  people = [ Person("Alice",    "Albany",   "AL"),
             Person("Bob",      "Boston", "MT"),
             Person("Charlie",  "Chico",  "IL"),
             Person("Gwen",     "Gaines",  "GA"),
             Person("James",    "Jordon",   "MT")]


  serialize_to_json(people)

  deserialized_people = deserialize_from_json()

  # Print the deserialized data
  for person in deserialized_people:
    print(f"Name: {person.name}, " +
          f"\t{person.city}, " +
          f"\t\t{person.state}")

main()

'''
JSON FILE
people.json
(Note the output below has been prettified for readablity)

[
{"name": "Alice", "city": "Albany", "state": "AL"},
{"name": "Bob", "city": "Boston", "state": "MT"},
{"name": "Charlie", "city": "Chico", "state": "IL"},
{"name": "Gwen", "city": "Gaines", "state": "GA"},
{"name": "James", "city": "Jordon", "state": "MT"}
]

 - - - - - - - - - - -
Output
Name: Alice,    Albany,         AL
Name: Bob,      Boston,         MT
Name: Charlie,  Chico,          IL
Name: Gwen,     Gaines,         GA
Name: James,    Jordon,         MT

'''
```

Code Summary

Person Class:

- Defines a `Person` class with attributes `name`, `city`, and `state`.

`serialize_to_json` Function:

- Takes a list of `Person` objects as input.
- Iterates through the list, creating dictionaries for each person with their attributes as key-value pairs.
- Creates a list `data` containing these dictionaries.
- Opens a file named "people.json" in write mode (`"w"`).
- Uses `json.dump` to convert the `data` list to a JSON string and write it to the file.

`deserialize_from_json` Function:

- Initializes an empty list `people` to store deserialized `Person` objects.
- Tries to open "people.json" in read mode (`"r"`).
- If successful, uses `json.load` to read the JSON data from the file and store it as a list in `data`.
- Iterates through the `data` list (containing dictionaries).
- For each dictionary, creates a new `Person` object using the values for "name", "city", and "state".
- Appends the created `Person` object to the `people` list.
- If the file is not found, it catches a `FileNotFoundError` and prints an error message.
- Finally, returns the list of deserialized `Person` objects.

`main` Function:

- Creates a sample list of `Person` objects with names, cities, and states.
- Calls `serialize_to_json` to convert this list to JSON and store it in "people.json".
- Calls `deserialize_from_json` to read the data back from "people.json" and get a list of deserialized `Person` objects.
- Prints the information of each deserialized person (name, city, state) in a formatted way.

0.25.5 Serialization - Pickle - Data

> **Danger**
> ONLY UNPICKLE FILES FROM A TRUSTED SOURCE. MALICOUS CODE CAN BE EXECUTED IN THE RECONSTRUCTION PROCESS

Pickle Data Demo Code

```
import pickle

# Define a class to represent a person with name, city, state
class Person:
  def __init__(self, name, city, state):
    self.name   = name
    self.city   = city
    self.state  = state

# Function to serialize a list of people to JSON
def serialize_to_pickle(people,filename):
  with open(filename, "wb") as f:
    pickle.dump(people, f)

# Function to deserialize JSON data back to a list of people
def deserialize_from_pickle(filename):
  people = []
  try:
    # open the file as read binary
    with open(filename, "rb") as f:
      people = pickle.load(f)
  except FileNotFoundError:
    print(f"Error: '{filename}' file not found.")
  return people

def main():
  # Example usage
  people = [ Person("Alice",    "Albany",   "AL"),
             Person("Bob",      "Boston", "MT"),
             Person("Charlie",  "Chico",  "IL"),
             Person("Gwen",     "Gaines",  "GA"),
             Person("James",    "Jordon",   "MT")]


  myfilename = "people.pickle"
  serialize_to_pickle(people, myfilename)

  deserialized_people = deserialize_from_pickle("people.pickle")

  # Print the deserialized data
  for person in deserialized_people:
    print(f"Name: {person.name}, " +
          f"\t{person.city}, " +
          f"\t\t{person.state}")

main()
```

0.25.6 Serialization - Pickle - Function

> **Danger**
> ONLY UNPICKLE FILES FROM A TRUSTED SOURCE. MALICOUS CODE CAN BE EXECUTED IN THE RECONSTRUCTION PROCESS

> **Caution**
> Pickling functions is done by reference and not the code itself. If other modules are involved, problems could arise. Also since it doesn't actually read the code, the **marshal** library must be used. Marshal.load will actually read the function code.

Pickle Function Demo Code

```
"""
Programmer: James Goudy

***
Pickling functions are NOT reccomended,
espcially if they require other modules.

Pickling functions does it by reference
and not the code itself.

To get around that, using the marshal
library (loads) will read the actual code.
***
"""

import pickle
import marshal
import types

def add(num1, num2):
  """
  Adds two numbers
  and returns the sum.
  """
  return num1 + num2

def pickle_add_func():

  # reads the actual add function code
  # and turns it into bytecode
  mybytecode = marshal.dumps(add.__code__)

  # write the bytecode to a binary file
  with open("add.pickle","wb") as outfile:
    pickle.dump(mybytecode,outfile)
  outfile.close()

  print("\nFunction Pickled")
```

```
def de_pickle_add_func():

  # read the file contents
  with open('add.pickle','rb') as infile:
    mypickledata = infile.read()
  infile.close()

  # unpickle contents to bytecode / binary
  mybytecode = pickle.loads(mypickledata)

  # get the code marshel bytecode object
  mycode = marshal.loads(mybytecode)

  # create the new function
  newAdd = types.FunctionType(mycode,globals())

  print("Function \'Depickled\'")
  ans = newAdd(10,30)

  print(ans)


def main():

  pickle_add_func()


  de_pickle_add_func()

  print("bye")

main()
```

0.25.7 Serialization - Pickle - Security Implications

Pickle, while convenient for serializing Python objects, comes with significant security risks. The core issue lies in its ability to reconstruct arbitrary code.

How Pickle Enables Code Execution

- **Object Reconstruction:**
  When unpickling, Python essentially executes code to recreate the original object.
- **Malicious Objects:**
  A carefully crafted pickle file can contain code disguised as object data.
  When unpickled, this code is executed, potentially giving an attacker control over the system.
- **Code Injection:** This vulnerability is often referred to as "pickle injection" or "unpickling vulnerabilities".

Attack Scenarios

- **Remote Code Execution (RCE):** An attacker can send a malicious pickle file to a vulnerable application, leading to arbitrary code execution on the target system.
- **Data Theft:** Malicious code within a pickle file can steal sensitive information.
- Denial of Service (DoS):
  A crafted pickle file can consume excessive resources, leading to a DoS attack.

Mitigating Risks

- **Never Unpickle Untrusted Data:** This is the most crucial rule. Only unpickle data from trusted sources.
- **Input Validation:** If you must unpickle user-provided data, implement strict validation to prevent malicious code injection.
- **Use Alternative Serialization Formats:** For data exchange, consider safer options like JSON or XML.
- **Consider Custom Picklers:**
  For specific use cases, you can create custom picklers with more control over the serialization process.
- **Security Audits:** Regularly review your code for potential pickle-related vulnerabilities.

Real-World Examples

- **Cloud Services:** Several cloud services have experienced security incidents due to pickle-related vulnerabilities.
- **Open-Source Projects:** Many open-source projects have had to address pickle-related security issues.

Conclusion

Pickle is a powerful tool for serializing Python objects, but its security implications cannot be ignored. By understanding the risks and implementing appropriate safeguards, you can significantly reduce the chances of exploitation.

0.25.8 Serialization HDF5

HDF5 (Hierarchical Data Format 5) is a file format designed to store and organize massive amounts of data in a structured and efficient way. It's particularly well-suited for scientific and engineering applications where data can be complex, heterogeneous, and immense.

Key Features of HDF5

- **Hierarchical structure:** Data can be organized into groups and subgroups, similar to directories in a file system.
- **Data types:** Supports a wide range of data types, including numerical arrays, strings, and even complex data structures.
- **Metadata:** Allows for embedding descriptive information about data, making it self-describing.
- **Large datasets:** Can handle datasets of virtually any size, efficiently storing and accessing data.
- **Compression:** Offers various compression methods to reduce file size without compromising data integrity.
- **Portability:** HDF5 files can be read and written on different platforms and operating systems.

Common Use Cases

- **Scientific data:** Storing experimental results, simulations, and images.
- **Image and signal processing:** Managing large image and signal datasets.
- **Bioinformatics:** Handling genomic and proteomic data.
- **Machine learning:** Storing and accessing training data.
- **Remote sensing:** Managing satellite and sensor data.

In essence, HDF5 provides a robust and flexible solution for managing and preserving large, complex datasets, making it a popular choice in various scientific and data-intensive fields.

Installing HDF5

```
pip install h5py numpy
```

Creating a Sample HDF5 File with People, Jobs, and Residencies

Understanding the Structure Before we dive into the code, let's outline the structure of our HDF5 file:

- **Root Group:** The top-level container.
- People Group:
  Contains information about individuals.
  - **Datasets:** Names, IDs, ages, etc.
  - **Attributes:** General information about the people group.
- Jobs Group:
  Contains information about jobs.
  - **Datasets:** Job titles, companies, salaries, etc.
  - **Attributes:** General information about the jobs group.
- Residencies Group:
  Contains information about residencies.
  - **Datasets:** City, state, zip code, etc.
  - **Attributes:** General information about the residencies group.

Python Code Using h5py Python

```
"""
HDF5 Code
Programmer: James Goudy

"""

import h5py
import numpy as np

def create_hdf5_file(filename):
  """Creates a sample HDF5 file with people, jobs, and residencies data.

  Args:
    filename: The name of the HDF5 file to create.
  """

  with h5py.File(filename, 'w') as hdf5_file:

    # Create groups - think of groups like folders
    people_group = hdf5_file.create_group('people')
    jobs_group   = hdf5_file.create_group('jobs')
    address_group = hdf5_file.create_group('address')

    # Datasets are like spreadsheets put into the folders
    # Create dataset - storing names, ids, and ages
    num_people = 5
    people_group.create_dataset('names', (num_people,), dtype='S20')
    people_group.create_dataset('ids', (num_people,), dtype='i')
    people_group.create_dataset('ages', (num_people,), dtype='i')

    # NOTE: in the examples below, the index position is the key that ties appropriate
    # data to the appropriate person.

    # load data into dataset
    people_group['names'][...] = np.array(['Alice', 'Bob', 'Charlie', 'David', 'Emily'],dtype=
    # people_group['names'][...] = np.array(['Alice', 'Bob', 'Charlie', 'David', 'Emily'],dtyp
    people_group['ids'][...] = np.array([1, 2, 3, 4, 5])
    people_group['ages'][...] = np.array([30, 25, 35, 40, 28])

    # create datasets - job title information
    jobs_group.create_dataset('jobTitles',(num_people), dtype='S20')

    # load data into dataset - alternative - assign in two steps
    myArrJobTitles = np.array(['Programmer','Cop','Baker','Pilot','Engineer'],dtype='S20')
    jobs_group['jobTitles'][...] = myArrJobTitles

    # Below is like the previous way to assign data to dataset
    # jobs_group['jobTitles'][...] = np.array(['Programmer','Cop','Baker','Pilot','Engineer'],

    # create datasets - address information
    address_group.create_dataset('cities',(num_people),dtype='S20')
    address_group.create_dataset('states',(num_people),dtype='S20')
```

```
    # load data
    address_group['cities'][...] = np.array(['Kali','Polson','Butte','Chicago','Whitefish'],dt
    address_group['states'][...] = np.array(['MT','CO','KS','IL','OH'],dtype='S20')


    # Add attributes / meta -data to the specific groups
    people_group.attrs['creation_date'] = "2024 -07 -25" # np.string_('2024 -07 -25')
    people_group.attrs['data_source'] = "Job Data"
    jobs_group.attrs['title_dir'] = "Job Title Directory"
    address_group.attrs['census_data'] = "Census Data"
    address_group.attrs['census_date'] = "Census Date"


    # Similar structure for jobs and residencies groups (omitted for brevity)

def read_hdf5_file(filename):
  """Reads the sample HDF5 file and displays the data.

  Args:
    filename: The name of the HDF5 file to read.
  """

  with h5py.File(filename, 'r') as hdf5_file:
    # Access groups
    # To decryppt we need to create the appropriate
    # folders to store information that is taken out of
    # the hdf5 file
    people_group = hdf5_file['people']
    jobs_group   = hdf5_file['jobs']
    address_group = hdf5_file['address']

    # Print people data
    # Note that the Names and Job Titles are
    # designated byte data.  This means that to
    # use them as strings, they need to be decode
    # from bytes to strings.
    print("People Data:")
    print("Names:", people_group['names'][...])
    print("IDs:", people_group['ids'][...])
    print("Ages:", people_group['ages'][...])
    print("Attributes:", people_group.attrs['creation_date'])
    print("Attributes:", people_group.attrs['data_source'])

    # Print jobs data (replace with actual code)
    print("Jobs Titles:",jobs_group['jobTitles'][...])
    # ...

    print("\nIterate through individual data\n - - - - -\n")
    for i in range(5):
      # note you can decode from either utf -8 or ascii
      print(f"{people_group['names'][i].decode('utf -8')}" + "\t" +
            str(people_group['ages'][i]) +  "\t"  +
```

```
            f"{jobs_group['jobTitles'][i].decode('utf -8')}" + "\n" +
            f"{address_group['cities'][i].decode('utf -8')}" + ", " +
            f"{address_group['states'][i].decode('ascii')}" + "\n")

    print("\n - - - - -\nSource Information / Attribution Data" + "\n" +
            "Creation date: " +
            people_group.attrs['creation_date'] +  "\n" +
            people_group.attrs['data_source']  + "\n" +
            jobs_group.attrs['title_dir'] + "\n" +
            address_group.attrs['census_data'] + "\n" +
            address_group.attrs['census_date'] + "\n")

if __name__ == '__main__':
  filename = 'sample.h5'
  create_hdf5_file(filename)

  print("\n\n - - - - - - - - - - -\n\n")

  read_hdf5_file(filename)

print("\nbye\n")



'''
OUTPUT

 - - - - - Input Data - - - - - -

 - - - - - Read Data From File - - - - - -

People Data:
Names: [b'Alice' b'Bob' b'Charlie' b'David' b'Emily']
IDs: [1 2 3 4 5]
Ages: [30 25 35 40 28]
Attributes: 2024 -07 -25
Attributes: Job Data
Jobs Titles: [b'Programmer' b'Cop' b'Baker' b'Pilot' b'Engineer']

Iterate through individual data
 - - - - -

Alice   30      Programmer
Kali, MT

Bob     25      Cop
Polson, CO

Charlie 35      Baker
Butte, KS

David   40      Pilot
Chicago, IL
```

```
Emily   28      Engineer
Whitefish, OH

 - - - - -
Source Information / Attribution Data
Creation date: 2024 -07 -25
Job Data
Job Title Directory
Census Data
Census Date

bye
'''
```

Code Summary

Purpose:

- Creates and manipulates an HDF5 file containing information about people, their jobs, and addresses.
- Demonstrates how to store, access, and read data from the HDF5 file.

Key Components:

- **HDF5 File:** A container for storing large amounts of data in a hierarchical structure.
- **Groups:** Represent folders within the HDF5 file for organizing data.
- **Datasets:** Similar to spreadsheets, storing specific data within groups.
- **Attributes:** Additional metadata associated with groups or datasets.

Code Functionality:

1. `create_hdf5_file(filename):`
    - Creates an HDF5 file with the specified filename.
    - Creates groups: `people`, `jobs`, and `address`.
    - Creates datasets within each group for storing names, IDs, ages, job titles, cities, and states.
    - Populates datasets with sample data.
    - Adds attributes to groups for metadata (creation date, data source, etc.).
2. `read_hdf5_file(filename):`
    - Opens the specified HDF5 file for reading.
    - Accesses the `people`, `jobs`, and `address` groups.
    - Prints data from the datasets, including names, IDs, ages, job titles, cities, and states.
    - Prints attributes associated with the groups.

Overall:

The code provides a basic example of how to use the `h5py` library to create and interact with HDF5 files. It demonstrates creating groups, datasets, and attributes, as well as reading and accessing data from the file.

0.25.9 Serialization - XML

XML: Extensible Markup Language

XML stands for ****Extensible Markup Language****. It's a text-based format for storing, transporting, and reconstructing data. Think of it as a way to structure information so that both humans and computers can understand it.

Key characteristics of XML:

- **Extensible:**
  You can create custom tags to describe your data. This makes it flexible for various applications.
- **Self-describing**:
  The tags provide information about the data, making it easier to understand without external documentation.
- **Platform-independent:**
  XML files can be read and processed on different operating systems and software.
- **Hierarchical:**
  Data is organized in a tree-like structure with parent and child elements.

How XML works: XML uses tags to enclose data. These tags define the structure of the data. For example:
XML

```
<book>
  <title>The Hitchhiker's Guide to the Galaxy</title>
  <author>Douglas Adams</author>
  <year>1979</year>
</book>
```

In this example, `book`, `title`, `author`, and `year` are tags. The text between the tags is the data.

Common uses of XML:

- **Data storage:**
  XML can store data in a structured format for later retrieval.
- **Data exchange:**
  It's used to transfer data between different systems and applications.
- **Configuration files:**
  Many software applications use XML to store configuration settings.
- **Web services:**
  XML is widely used for exchanging data between web services.

Example code

```
,,,
Programmer: James Goudy
Project: XML


you must know the xml schema

library
    book
        title
        author
        year
```

```
,,,

import xml.etree.ElementTree as ET

def create_xml_file1(filename):
    """Creates a sample XML file with book information."""
    root = ET.Element("library")
    book1 = ET.SubElement(root, "book")
    ET.SubElement(book1, "title").text = "The Hitchhiker's Guide to the Galaxy"
    ET.SubElement(book1, "author").text = "Douglas Adams"
    ET.SubElement(book1, "year").text = "1979"

    book2 = ET.SubElement(root, "book")
    ET.SubElement(book2, "title").text = "The Lord of the Rings"
    ET.SubElement(book2, "author").text = "J.R.R. Tolkien"
    ET.SubElement(book2, "year").text = "1954"

    tree = ET.ElementTree(root)
    tree.write(filename)

def create_xml_file2(filename,BookArray,theSchmea):

    # passing in the xml schema

    root = ET.Element(theSchmea[0])

    # iterate through the book information
    for book in BookArray:
        abook = ET.SubElement(root,theSchmea[1])
        ET.SubElement(abook,theSchmea[2]).text = book[theSchmea[2]]
        ET.SubElement(abook,theSchmea[3]).text = book[theSchmea[3]]
        ET.SubElement(abook,theSchmea[4]).text = str(book[theSchmea[4]])


    tree = ET.ElementTree(root)
    tree.write(filename)


def read_xml_file(filename):
    print("\nRead File  " + filename)
    """Reads the created XML file and prints book information."""
    tree = ET.parse(filename)
    # tree = ET.parse("zbooks1.xml")
    root = tree.getroot()


    # for book in root.iter("book"):
    for book in root:

        title = book.find("title").text
        author = book.find("author").text
        year = book.find("year").text
```

```
        print(f"Title: {title}")
        print(f"Author: {author}")
        print(f"Year: {year}")
        print()



if __name__ == "__main__":


    BookSchema = ["libary","book","title","author","year"]

    books = [
    {"title": "Introduction to Algorithms", "author": "Cormen", "year": 1989},
    {"title": "Clean Code", "author": "Martin", "year": 2012},
    {"title": "Structure and Interpretation of Computer Programs", "author": "Sussman", "year"
    ]

    history_books = [
    {"title": "Sapiens: A Brief History of Humankind", "author": "Yuval Noah Harari", "year":
    {"title": "Guns, Germs, and Steel: The Fates of Human Societies", "author":"Diamond", "yea
    {"title": "A Short History of Nearly Everything", "author": "Bryson", "year": 2003},
    {"title": "1491: New Revelations of the Americas Before Columbus", "author": "Mann", "year
    {"title": "The Rise and Fall of the Roman Empire", "author": "Gibbon", "year": 1776}
]

    # write xml files

    zthefilename1 = "zbooks1.xml"
    zthefilename2 = "zbooks2.xml"

    create_xml_file1(zthefilename1)
    create_xml_file2(zthefilename2,history_books,BookSchema)

    # Read the files
    read_xml_file(zthefilename1)
    print("\n - - - - - - - - part 2 - - - - - - - - - -\n")
    read_xml_file(zthefilename2)

    print("\nbye\n")

"""
NOTE: The output has been prettified.
 - - - zbooks1.xml - - -
<library>
   <book>
      <title>The Hitchhiker's Guide to the Galaxy</title>
      <author>Douglas Adams</author>
      <year>1979</year>
   </book>
   <book>
      <title>The Lord of the Rings</title>
      <author>J.R.R. Tolkien</author>
```

```
      <year>1954</year>
   </book>
</library>

 - - - zbooks2.xml - - -
<libary>
 <book>
  <title>Sapiens: A Brief History of Humankind</title>
  <author>Yuval Noah Harari</author>
  <year>2011</year>
 </book>
 <book>
  <title>Guns, Germs, and Steel: The Fates of Human Societies</title>
  <author>Jared Diamond</author>
  <year>1997</year>
 </book>
 <book>
  <title>A Short History of Nearly Everything</title>
  <author>Bill Bryson</author>
  <year>2003</year>
 </book>
 <book>
  <title>1491: New Revelations of the Americas Before Columbus</title>
  <author>Charles C. Mann</author>
  <year>2005</year>
 </book>
 <book>
  <title>The Rise and Fall of the Roman Empire</title>
  <author>Edward Gibbon</author>
  <year>1776</year>
 </book>
</libary>


 - - - Program Output - - -

Read File  zbooks1.xml
Title: The Hitchhiker's Guide to the Galaxy
Author: Douglas Adams
Year: 1979

Title: The Lord of the Rings
Author: J.R.R. Tolkien
Year: 1954


 - - - - - - - - part 2 - - - - - - - - - -


Read File  zbooks2.xml
Title: Sapiens: A Brief History of Humankind
Author: Yuval Noah Harari
Year: 2011
```

```
Title: Guns, Germs, and Steel: The Fates of Human Societies
Author: Diamond
Year: 1997

Title: A Short History of Nearly Everything
Author: Bryson
Year: 2003

Title: 1491: New Revelations of the Americas Before Columbus
Author: Mann
Year: 2005

Title: The Rise and Fall of the Roman Empire
Author: Gibbon
Year: 1776


bye


"""
```

Program Summary

Purpose:

- Creates XML files containing book information.
- Reads and displays the contents of these XML files.

Key Components:

1. **XML Schema:** Defines the structure of the XML data, including elements like `library`, `book`, `title`, `author`, and `year`.
2. **Book Data:** Stores book information in arrays of dictionaries, with each dictionary representing a book and its details.
3. XML Creation Functions:
   - `create_xml_file1`: Creates a hardcoded XML file with two books.
   - `create_xml_file2`: Creates an XML file based on a given book array and XML schema.
4. XML Reading Function:
   - `read_xml_file`: Parses an XML file, extracts book information, and prints it to the console.

Overall Flow:

1. Defines the XML schema and book data arrays.
2. Creates two XML files using the defined functions.
3. Reads and displays the contents of both XML files.

Key Points:

- The program demonstrates how to create and read XML files in Python using the `xml.etree.ElementTree` module.
- It highlights the importance of an XML schema for defining data structure.
- The code provides a basic example of working with XML data.

0.25.10 Serialization - YAML

YAML stands for **YAML Ain't Markup Language** (originally Yet Another Markup Language). It's a data serialization language designed to be human-readable and easy to understand. Think of it as a way to store structured information in a text file, similar to JSON or XML, but with a more intuitive syntax.

How is it used?

YAML files are commonly used for:

- Configuration files:
  YAML's readability makes it ideal for storing application settings, database connections, and other configuration parameters.
- Data serialization:
  You can use YAML to store complex data structures like lists, dictionaries, and objects in a portable format.
- Document formats:
  YAML can be used to represent structured documents, such as those used in content management systems or data exchange formats.

Key advantages of YAML:

- Readability:
  YAML's syntax is designed to be human-friendly, making it easy to write, read, and modify.
- Flexibility:
  It supports various data types, including scalars, sequences, mappings, and more complex structures.
- Simplicity:
  YAML's syntax is relatively straightforward compared to other serialization formats.

Python Example: Writing and Reading a YAML File

Prerequisites:

- Python installed
- `pyyaml` library installed (`pip install pyyaml`)

Python

```
,,,
Programmer: James Goudy
Project: yaml demo

,,,

import yaml

def write_yaml_file(data, filename):
  """Writes data to a YAML file.

  Args:
    data: The data to be written.
    filename: The name of the YAML file.
  """

  with open(filename, 'w') as yaml_file:
    yaml_file.write('# Demographics')
    yaml.dump(data, yaml_file, default_flow_style=False)
```

```
    yaml_file.close()

  # NOTE: if more items are created in stages
  # calling the file the next time to add items,
  # use the 'a' attribute. 'w' first time, 'a' next time on

def read_yaml_file(filename):
  """Reads data from a YAML file.

  Args:
    filename: The name of the YAML file.

  Returns:
    The data loaded from the YAML file.
  """
  with open(filename, 'r') as yaml_file:
    data = yaml.safe_load(yaml_file)
    yaml_file.close()
  return data



def main():
  # creating embedded dictionaries
  uk_c = {'UK':['London','Bath','York']}
  fr_c = {'FR':['Paris','Lyon','Nice','Metz']}
  travel_c = {'travel' : [uk_c,fr_c,'GR']}

  data = {
    'name': 'Jimbo',
    'age': 30,
    'city': 'New York',
    'hobbies': ['reading', 'coding', travel_c,'cooking']
  }

  # Write data to a YAML file
  yaml_file = 'data.yaml'
  write_yaml_file(data, yaml_file)

  # Read data from the YAML file
  loaded_data = read_yaml_file(yaml_file)

  print('\nAll Data\nNote the embedded dictionaries')
  print(loaded_data)

#   show the progression on how
#   to access a specific city
  print('\nAll hobbies')
  td = loaded_data['hobbies']
  print(td)

  print('\nAll Travel Data')
  td = loaded_data['hobbies'][2]
```

```
  print(td)

  print('\nAll Travel Countries')
  td = loaded_data['hobbies'][2]['travel']
  print(td)

  print('\nAll French Items (dictionary)')
  td = loaded_data['hobbies'][2]['travel'][1]
  print(td)

  print('\nAll French Cities')
  td = loaded_data['hobbies'][2]['travel'][1]['FR']
  print(td)

  print('\nFrench City of Lyon')
  td = loaded_data['hobbies'][2]['travel'][1]['FR'][1]
  print(td)

  print('\n yaml bye\n')

main()

'''
Output file

# Demographicsage: 30
city: New York
hobbies:
 - reading
 - coding
 - travel:
  - UK:
    - London
    - Bath
    - York
  - FR:
    - Paris
    - Lyon
    - Nice
    - Metz
  - GR
 - cooking
name: Jimbo
'''
```

Explanation:

This Python program demonstrates how to use the `yaml` library to read and write data to a YAML file.

Key functionalities:

- Defines two functions:
  - `write_yaml_file`: Writes data to a YAML file in a human-readable format.
  - `read_yaml_file`: Reads data from a YAML file and returns it as a Python object.

- **Creates a sample data structure:** A Python dictionary containing various data types, including nested dictionaries.
- **Writes the data to a YAML file:** Uses the `write_yaml_file` function to create a YAML file named 'data.yaml'.
- **Reads the data from the YAML file:** Uses the `read_yaml_file` function to load the YAML data into a Python object.
- **Demonstrates data access:** Shows how to access different parts of the loaded data, including nested dictionaries, to illustrate how to work with YAML data in Python.

0.26 Software Testing

0.26.1 Software Testing Concepts

Unit Testing

- **Definition:** Unit testing involves testing individual components or units of code in isolation. These units could be functions, methods, or classes.
- **Purpose:** To verify that each unit works as expected and meets its specific requirements.
- Benefits:
  - Early detection of defects
  - Improved code quality
  - Increased code maintainability
  - Supports regression testing

Integration Testing

- **Definition:** Integration testing combines multiple units or components to test their interactions and communication.
- **Purpose:** To verify that different parts of the software work together as expected.
- Benefits:
  - Identifies interface-related defects
  - Ensures data integrity and flow
  - Improves system stability

Regression Testing

- **Definition:** Regression testing is the process of re-executing a subset of tests to ensure that previously developed and tested software still performs correctly after changes have been made.
- **Purpose:** To prevent new code from introducing unintended side effects or breaking existing functionality.
- Benefits:
  - Maintains software quality over time
  - Reduces the risk of introducing new defects
  - Improves confidence in software changes

In essence:

- Unit testing focuses on the smallest parts of the code.
- Integration testing focuses on how those parts work together.
- Regression testing focuses on ensuring that previous functionality remains intact after changes.

0.26.2 Software Testing - Pytest

Pytest is a powerful and flexible Python testing framework that simplifies the process of writing and running tests. It offers a clean syntax, rich features, and extensive plugin support, making it a popular choice for testing Python applications.

> **Note**
>
> Pytest needs to be installed.

```
pip install pytest
```

Basic Example

Let's start with a simple example:
Python

```
import pytest

def add(x, y):
    return x + y

def test_add():
    assert add(2, 3) == 5
```

- ***Test File Naming:*** Pytest automatically discovers test files and functions. Test files typically start with `test_` or end with `_test.py`.
- **Test Function Naming:** Test functions usually start with `test_`.
- **Assertion:** The `assert` statement checks if the actual result matches the expected result.

```
"""
# Below assumes a windows machine
# MUST CALL FROM THE COMMAND LINE:

pytest pytesting_example1.py/ -v

# Writes to a text file
pytest pytesting_example1.py/ -v > outputfilename.txt

# Display text file
type outputfilename.txt


"""
import pytest


def add(x, y):
    return x + y

def test_1():

    result= add(2,3)
    assert result == 5
```

```
    result= add(1,4)
    assert result == 5

    result= add(8, -3)
    assert result == 5

def test_2():
    result= add(2,3)
    assert result == 5

    result= add(1,4)
    assert result == 5

    # I'm asserting the answer is 5
    # when the reality is 41
    # as written will cause a pytest error
    result= add(1,40)
    assert result == 5

    result= add(8, -3)
    assert result == 5

def test_3():
    result= add(2,3)
    assert result == 5

    result= add(1,4)
    assert result == 5

if __name__ == '__main__':

    test_1()
    test_2()
    test_3()
    print('\n\nbye\n')


"""
OUTPUT
============================= test session starts ==============================
platform win32 - - Python 3.12.4, pytest -8.3.2, pluggy -1.5.0 - - C:\Program Files\Python312\
cachedir: .pytest_cache
rootdir: E:\My Drive\0_CSCI_127_JBD\Teaching_Modules\Module_Testing
collecting ... collected 3 items

pytesting_example1.py::test_1 PASSED                                     [ 33%]
pytesting_example1.py::test_2 FAILED                                     [ 66%]
pytesting_example1.py::test_3 PASSED                                     [100%]

================================== FAILURES ===================================
_________________________________ test_2 _____________________________________
```

```
    def test_2():
        result= add(2,3)
        assert result == 5

        result= add(1,4)
        assert result == 5

        result= add(1,40)
>       assert result == 5
E       assert 41 == 5

pytesting_example1.py:47: AssertionError
============================== warnings summary ===============================
pytesting_example1.py:73
  E:\My Drive\0_CSCI_127_JBD\Teaching_Modules\Module_Testing\pytesting_example1.py:73: SyntaxW
    """

 - - Docs: https://docs.pytest.org/en/stable/how -to/capture -warnings.html
=========================== short test summary info ===========================
FAILED pytesting_example1.py::test_2 - assert 41 == 5
=================== 1 failed, 2 passed, 1 warning in 0.11s ====================
"""
```

Parametrized Tests

To test a function with different inputs, use `pytest.mark.parametrize`:

Python

```
import pytest

def add(x, y):
    return x + y

@pytest.mark.parametrize("x, y, expected", [
    (2, 3, 5),
    ( -1, 1, 0),
    (0, 0, 0)
])
def test_add(x, y, expected):
    assert add(x, y) == expected
```

Fixtures

Fixtures are functions that provide setup and teardown for tests. They help manage shared resources:

Python

```
import pytest

@pytest.fixture
def my_fixture():
    # Setup code
    return some_value

def test_something(my_fixture):
    # Use the fixture
```

```
    assert my_fixture == expected_value
```

Test Classes

You can organize tests into classes using `pytest.mark.usefixtures`:
Python

```
import pytest

class TestClass:
    @pytest.fixture(autouse=True)
    def setup(self):
        print("Setting up")

    def test_one(self):
        assert True

    def test_two(self):
        assert False
```

Additional Features

- **Skipping Tests:** `pytest.skip` or `pytest.xfail` to skip tests conditionally.
- **Grouping Tests:** Use `pytest.mark` for custom markers to group tests.
- **Capturing Output:** Use `capsys` fixture to capture stdout and stderr.
- ***Monkeypatching:*** Modify modules or functions for testing purposes.
- ***Plugins:*** Extend pytest functionality with plugins.

Running Tests:
You can run tests from the command line:
Cmd

```
pytest
```

Or specify a test file or directory:
Cmd

```
pytest test_module.py
```

Key Points:

- Pytest is easy to learn and use.
- It offers a rich feature set for various testing scenarios.
- It integrates well with other Python tools and libraries.

By understanding these basics and exploring the extensive documentation, you can effectively use Pytest to write comprehensive and maintainable tests for your Python projects.

Would you like to explore a specific feature or use case in more detail?

0.26.3 Software Testing Unit Testing

Unit Testing

- **Definition:** Unit testing involves testing individual components or units of code in isolation. These units could be functions, methods, or classes.
- **Purpose:** To verify that each unit works as expected and meets its specific requirements.
- Benefits:
  - Early detection of defects
  - Improved code quality
  - Increased code maintainability
  - Supports regression testing

Integration Testing

- **Definition:** Integration testing combines multiple units or components to test their interactions and communication.
- **Purpose:** To verify that different parts of the software work together as expected.
- Benefits:
  - Identifies interface-related defects
  - Ensures data integrity and flow
  - Improves system stability

UnitTest

Unittest is Python's standard library for writing and running unit tests. It provides a structured approach to testing individual units of code (functions, methods, classes) to ensure they behave as expected.

https://docs.python.org/3/library/unittest.html

Key Features:

- **Test Cases:** Define individual test scenarios using the `TestCase` class.
- **Test Suites:** Organize test cases into collections for efficient execution.
- **Assertions:** Verify expected outcomes using methods like `assertEqual`, `assertTrue`, etc.
- **Fixtures:** Set up and tear down test environments using `setUp()` and `tearDown()` methods.

UnitTest Examples

Basic Example: Testing a Simple Function Python

```
"""
Unit Testing is for the testing of functions
and methods
"""

import unittest

def add(x, y):
    return x + y

class TestAdd(unittest.TestCase):


    def test_add_positive_true(self):
        result = add(2, 3)
```

```
        self.assertEqual(result, 5)

    def test_add_positive_true_bad(self):
        result = add(2, 3)
        self.assertEqual(result, 100)

    def test_add_negative(self):
        result = add( -1, 2)
        self.assertEqual(result, 1)

# - - - - - - Examples of comparitive values

    def test_3_gt_2(self):
        self.assertGreater(3,2);

    def test_2_gt_3(self):
        self.assertGreater(2,3);

if __name__ == '__main__':
    # Setting the verbosity to 2
    # Gives details on what tests
    # were ran and the outcomes
    unittest.main(verbosity=2)


"""
output:


test_2_gt_3 (__main__.TestAdd.test_2_gt_3) ... FAIL
test_3_gt_2 (__main__.TestAdd.test_3_gt_2) ... ok
test_add_negative (__main__.TestAdd.test_add_negative) ... ok
test_add_positive_true (__main__.TestAdd.test_add_positive_true) ... ok
test_add_positive_true_bad (__main__.TestAdd.test_add_positive_true_bad) ... FAIL

======================================================================
FAIL: test_2_gt_3 (__main__.TestAdd.test_2_gt_3)
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Traceback (most recent call last):
  File "e:\My Drive\0_CSCI_127_JBD\Teaching_Modules\Module_Testing\unitTesting_example1.py", l
    self.assertGreater(2,3);
    ^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 2 not greater than 3

======================================================================
FAIL: test_add_positive_true_bad (__main__.TestAdd.test_add_positive_true_bad)
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Traceback (most recent call last):
  File "e:\My Drive\0_CSCI_127_JBD\Teaching_Modules\Module_Testing\unitTesting_example1.py", l
    self.assertEqual(result, 100)
AssertionError: 5 != 100

 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
Ran 5 tests in 0.004s

FAILED (failures=2)

"""
```

Testing a Class Method Python

```
"""
Unit Testing is for the testing of functions
and methods
"""

import unittest


class AddCaclulator:

    def __init__(self) -> None:
        pass

    def add(x, y):
        return x + y

class TestAdd(unittest.TestCase):

    def test_add_positive_true(self):
        result = myCalculator.add(2, 3)
        self.assertEqual(result, 5)

    def test_add_positive_true_bad(self):
        result = myCalculator.add(2, 3)
        self.assertEqual(result, 100)

    def test_add_negative(self):
        result = myCalculator.add( -1, 2)
        self.assertEqual(result, 1)

    # - - - - - - Examples of comparitive values
    def test_3_gt_2(self):
        self.assertGreater(3,2);

    def test_2_gt_3(self):
        self.assertGreater(2,3);

# global object
myCalculator = AddCaclulator

if __name__ == '__main__':
    # Setting the verbosity to 2
    # Gives details on what tests
    # were ran and the outcomes
    unittest.main(verbosity=2)
```

```
,,,
Output
# - - - - - - - - - - - - - - - - - - - -
test_2_gt_3 (__main__.TestAdd.test_2_gt_3) ... FAIL
test_3_gt_2 (__main__.TestAdd.test_3_gt_2) ... ok
test_add_negative (__main__.TestAdd.test_add_negative) ... ok
test_add_positive_true (__main__.TestAdd.test_add_positive_true)
... ok
test_add_positive_true_bad (__main__.TestAdd.test_add_positive_true_bad) ... FAIL

======================================================================
FAIL: test_2_gt_3 (__main__.TestAdd.test_2_gt_3)
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Traceback (most recent call last):
  File "e:\My Drive\0_CSCI_127_JBD\Teaching_Modules\Module_Testing\unitTesting_example2.py", l
    self.assertGreater(2,3);
    ^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 2 not greater than 3

======================================================================
FAIL: test_add_positive_true_bad (__main__.TestAdd.test_add_positive_true_bad)
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Traceback (most recent call last):
  File "e:\My Drive\0_CSCI_127_JBD\Teaching_Modules\Module_Testing\unitTesting_example2.py", l
    self.assertEqual(result, 100)
AssertionError: 5 != 100

 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Ran 5 tests in 0.004s

FAILED (failures=2)

,,,
```

Using Assertions Unittest provides several assertion methods to check different conditions:

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`
There are also other methods used to perform more specific checks, such as:	
`assertAlmostEqual(a, b)`	`round(a-b, 7) == 0`
`assertNotAlmostEqual(a, b)`	`round(a-b, 7) != 0`
`assertGreater(a, b)`	`a >b`
`assertGreaterEqual(a, b)`	`a >= b`
`assertLess(a, b)`	`a <b`
`assertLessEqual(a, b)`	`a <= b`
`assertRegex(s, r)`	`r.search(s)`
`assertNotRegex(s, r)`	`not r.search(s)`
`assertCountEqual(a, b)`	a and b have the same elements in the same number, regardless of their order.

Test Fixtures

Unittest provides `setUp()` and `tearDown()` methods to set up and clean up resources before and after each test case:

Python

```
import unittest

class TestDatabase(unittest.TestCase):
    def setUp(self):
        # Connect to database

    def tearDown(self):
        # Close database connection

    def test_query(self):
        # Perform database query

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

Running Tests To run the tests, you can use the following command in your terminal:

Cmd

```
python test_file.py
```

For more detailed output, use the `-v` flag:

Cmd

```
python -m unittest -v test_file.py
```

Remember:

- Each test case should be independent.
- Test cases should cover different scenarios and edge cases.
- Use clear and descriptive test case names.
- Strive for high test coverage.

0.26.4 Software Testing - Unittest vs Pytest

Unittest and ***Pytest*** are both testing frameworks used in Python for writing and running tests. They help ensure code quality and reliability.

Unittest

- ***Built-in:*** It's part of Python's standard library, so no additional installation is required.
- **Structure:** Uses a class-based structure with methods for test cases.
- ***Verbosity:*** Can be more verbose compared to Pytest.
- ***Features:*** Provides basic test discovery, fixture support, and test suites.

Pytest

- **Third-party:** Needs to be installed separately.
- **Simplicity:** Emphasizes simplicity and readability with a more concise syntax.
- ***Flexibility:*** Offers powerful features like fixtures, parameterization, and test discovery.
- ***Plugins:*** Has a rich ecosystem of plugins for extending functionality.

Key Differences

Feature	Unittest	Pytest
Installation	Built-in	Requires installation
Syntax	Class-based	Function-based
Verbosity	More verbose	Concise
Features	Basic test discovery, fixtures, test suites	Powerful fixtures, parameterization, test discovery, plugins

When to Use Which?

- **Unittest:** Suitable for projects that prefer a structured approach and are already using the standard library extensively.
- **Pytest:** Ideal for projects that prioritize simplicity, flexibility, and a large test suite. It's often preferred for its ease of use and extensive features.

In summary, both Unittest and Pytest are effective for testing Python code. The choice often depends on project requirements, team preferences, and the complexity of the test suite.

0.27 Miscellany

0.27.1 Built-in Variable ___name___

The Python variable `__name__` is a special built-in variable that is a part of the Python runtime environment. Here's what it represents:

- When a Python script is executed, the interpreter reads the source file and defines a few special variables before executing the code. One of these special variables is `__name__`.
- If the script is being run as the main program, the value of `__name__` is set to `'__main__'`. This indicates that the script is not being imported from another module but is being executed directly.
- Conversely, if the script is being imported into another module, **name** is set to the name of the script/module being imported.

This behavior of the `__name__` variable is useful for determining whether a script is being run on its own or being used by another script as a module. It allows developers to control which parts of their code should be executed when the module is run directly and which parts should be executed when the module is imported. **Bold** the relevant parts of the response to improve readability.

Here's what it does:

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

- `__name__` is a special built-in variable in Python that represents the name of the current module.
- When a Python script is run directly (for example, `python script.py` from the command line), the `__name__` variable is set to `"__main__"`.
- If the script is being imported into another module, `__name__` will be set to the name of the script/module.
- `main()` is typically a function that contains the code to be executed when the script is run directly. It's not a built-in function but is often defined by the programmer to organize code.

So, the `if` statement checks: if this script is the main program being executed, then call the `main()` function and run the code inside it. This allows a script to provide functionality for other scripts to import, while also providing a standalone functionality when the script is executed on its own. It's a way to make a Python file both reusable as a module and executable as a script. **Bold** the relevant parts of the response to improve readability.

Demo Code

Here's a short Python program demonstrating the `__name__` attribute:

Python

```
def greet(name):
  print(f"Hello, {name}!")

if __name__ == "__main__":
  greet("World")  # Call greet only when running as a script

print("This line is always executed")
```

Explanation:

1. We define a function `greet` that takes a name and prints a greeting.
2. The `if __name__ == "__main__":` block checks if the script is being run directly (as the main program) or imported as a module.
3. Inside the `if` block, we call `greet("World")`. This ensures the greeting is printed only when the script is executed directly.

4. Outside the `if` block, we print another message. This line will always be executed, regardless of whether the script is run directly or imported.

Running the program:

- Save the code as a Python file (e.g., `demo.py`).
- To run the script directly, open a terminal and navigate to the directory where you saved the file. Then, type:

Bash

```
python demo.py
```

This will print:

```
Hello, World!
This line is always executed
```

- To import the script as a module, create another Python file and use the `import` statement:

Python

```
import demo

demo.greet("Friend")  # Call the greet function from the imported module
```

This will only print:

```
Hello, Friend!
```

The `__name__` check ensures the greeting functionality is only executed when running the script directly, not when imported as a module.

> In Short: It Allows You to Execute Code When the File Runs as a Script, but Not When It's Imported as a Module

0.27.2 The Zen of Python,

by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

0.27.3 Python Application Packaging Tools

Overview, Pricing, and URLs

This document presents a consolidated overview of major Python packaging tools, their costs, licenses, URLs, and practical notes suitable for classroom, textbook, and professional use.

---

1. Overview of Packaging Tools

Below is a summary of the most widely used tools for converting Python applications (including GUIs) into standalone executables or native installers.

cx_Freeze A free, open-source tool that works across Windows, macOS, and Linux. It bundles a Python script and its dependencies into a distributable executable directory. Very stable and ideal for teaching environments.
URL: https://pypi.org/project/cx-Freeze/

Nuitka A Python-to-C compiler that generates optimized binaries. It offers both free and commercial versions. The free version is suitable for typical course assignments, while the paid versions add advanced features.
URL: `https://nuitka.net/`

Briefcase (BeeWare Project) Creates native installers and application bundles for desktop and mobile platforms. Fully open-source. Excellent when you want applications that feel platform-native.
URL: https://beeware.org/briefcase/

---

py2exe A simple, Windows-only packaging solution. Useful for classrooms or labs where Windows is the standard environment.
URL: https://www.py2exe.org/

---

py2app The macOS equivalent of py2exe. Packages Python applications into macOS .app bundles.
URL: https://py2app.readthedocs.io/en/latest/

---

2. Pricing & Licensing Table

Tool	Price / License	Notes
cx_Freeze	Free, open source (PSF-derived license)	Ideal for teaching and student projects. Requires small license snippet in output per documentation.
py2exe	Free, open source (MIT / MPL / X11)	Windows-only; very straightforward.
py2app	Free, open source	Mac-only; simple and stable.
Nuitka	Free core (Apache), Paid tiers: 250/yr, 400/yr, 1000/yr	Free version works for most student needs; paid tiers give extended features and support.
Briefcase	Free, open source (BSD-3-Clause)	Produces native installers; distribution to app stores may require respective account fees.

---

3. When to Use Each Tool

- **cx_Freeze**: Best all-around free tool for desktop apps across platforms.
- **py2exe / py2app**: Great when students are working on a single OS environment.
- **Nuitka**: Use when you want compiled performance or to demonstrate how Python can become real binaries.
- **Briefcase**: Best for polished, platform-native installers and mobile distribution.

---

4. Additional Notes

- Packaging tools help students understand software distribution beyond the source-code level.
- Make sure students test executables on a fresh machine when possible.
- If building GUI apps (Tkinter, PyQt, PySide, Kivy), choose a packager that supports non-Python assets.
- Packaging is an excellent appendix or final chapter for a programming or GUI development textbook.

0.27.4 Compile Python To Executable - PyInstaller

PyInstaller is a free and open-source program that lets you bundle a Python application and all its dependencies into a single package. This package can then be easily distributed and run by users without requiring them to have Python or any of its libraries installed on their systems.

How does it work?

- **Analyze your Python script:** PyInstaller reads your Python script and identifies all the modules and libraries that it needs to run.
- **Collect dependencies:** PyInstaller then collects copies of all those modules and libraries. This includes not only the Python standard library but also any third-party libraries that your script uses.
- **Package everything up:** Finally, PyInstaller packages all of the files together into a single executable file or a folder containing an executable file and its supporting files. This package can be run on any system that has the same operating system as the one where it was created.

Benefits of using PyInstaller:

- **Easy to distribute:** You can easily share your Python applications with others without worrying about whether they have the right dependencies installed.
- **No Python installation required:** Users don't need to have Python installed on their systems to run your application.
- **Standalone:** Your application is self-contained and doesn't rely on any external files or libraries.
- **Secure:** You can control exactly which libraries are included in your application, which can help to improve security.

Overall, PyInstaller is a powerful tool that can make it easy to distribute and run your Python applications.

Here are some additional resources that you may find helpful:

- **PyInstaller website:** https://www.pyinstaller.org/
- **PyInstaller documentation:** https://www.pyinstaller.org/
- Website: PyInstaller Manual
- https://stackoverflow.com/questions/5458048/how-can-i-make-a-python-script-standalone-executable-to-run-without-any-dependen

(JGAI)

Quick Start

Install PyInstaller from PyPI:

```
pip install pyinstaller
```

Go to your program's directory and run:

```
pyinstaller yourprogram.py
```

This will generate the bundle in a subdirectory called `dist`.

```
pyinstaller -F yourprogram.py
```

Adding -F (or –onefile) parameter will pack everything into single "exe".

```
pyinstaller -F - -paths=<your_path>\Lib\site -packages  yourprogram.py
```

running into "ImportError" you might consider side-packages.

```
pip install pynput==1.6.8
```

still running into an Import-Erorr - try to downgrade pyinstaller - see Getting error when using pynput with pyinstaller

For a more detailed walkthrough, see the manual.

0.27.5 PIP Python Package Installer

pip is a package manager for Python. It's a command-line tool that allows you to install, uninstall, and manage Python packages (libraries).

Key functions of pip:

- **Installation:** Downloads and installs Python packages from the Python Package Index (PyPI) or other repositories.
- **Uninstallation:** Removes installed packages from your system.
- **Listing:** Displays a list of installed packages.
- **Search:** Finds packages on PyPI based on keywords.
- **Update:** Checks for and installs newer versions of installed packages.

Example usage:
To install the NumPy package:
Cmd/Powershell

```
pip install numpy
```

To uninstall the NumPy package:
Cmd/Powershell

```
pip uninstall numpy
```

To list all installed packages:
Cmd/Powershell

```
pip list
```

Note: Most Python distributions (like Anaconda or Miniconda) come with pip pre-installed. If you're using a different distribution or need to install pip manually, you can follow the instructions on the official Python website.

Major pip Commands

Here are some of the most common pip commands you'll likely encounter:

Installation and Uninstallation

- `pip install <package_name>`:
  Installs a specific package from PyPI.
  - Example: `pip install numpy`
- **`pip install -r requirements.txt`:** Installs all packages listed in a `requirements.txt` file.
- `pip uninstall <package_name>`:
  Uninstalls a specific package.
  - Example: `pip uninstall requests`

Package Management

- **`pip list`:** Lists all installed packages.
- **`pip show <package_name>`:** Displays detailed information about a package.
- **`pip search <keyword>`:** Searches for packages on PyPI based on a keyword.
- **`pip freeze`:** Generates a list of installed packages and their versions, often used to create a `requirements.txt` file.

Updating Packages

- **`pip install -upgrade <package_name>`:** Upgrades a specific package to the latest version.
- **`pip install -upgrade -r requirements.txt`:** Upgrades all packages listed in a `requirements.txt` file.

Virtual Environments

- **`pip install virtualenv`:** Installs the `virtualenv` package to create isolated Python environments.
- **`virtualenv <env_name>`:** Creates a new virtual environment.
- **`source <env_name>/bin/activate`:** Activates the virtual environment.
- **`deactivate`:** Deactivates the current virtual environment.

Additional Notes:

- You can use the `-q` flag to suppress output.
- For more advanced usage, refer to the official pip documentation: https://pypi.org/project/python-pip/

0.27.6 Python Environments

Python environments are isolated spaces where you can install and use different versions of Python and its packages without affecting your system's default Python installation. This is crucial for managing multiple projects that require different dependencies or versions of Python.

Why Use Python Environments?

- **Dependency Management:** Avoid conflicts between packages required by different projects.
- **Version Control:** Easily switch between different Python versions for compatibility testing or experimental development.
- **Project Isolation:** Keep project-specific packages and configurations separate from your system's default Python.
- **Collaboration:** Share projects with others without affecting their existing environments.

Popular Environment Managers:

- **Virtualenv:** A classic tool for creating and managing virtual environments.
- **venv:** The built-in environment management tool in Python 3.3 and later.
- **conda:** A versatile package and environment manager that can handle both Python and other languages.

Creating and Activating Environments:

1. **Choose an environment manager:** Select the one that best suits your needs and preferences.
2. **Create a new environment:** Use the appropriate command to create a new environment with a specific Python version or package set.
3. **Activate the environment:** Make the newly created environment active, so your Python interpreter and package manager use its settings.

Example (using venv):
Cmd /Powershell

```
# Create a new environment named "myproject" with Python
python -m venv myproject

# Activate the environment
# source myproject/bin/activate
# On Windows, use
myproject\Scripts\activate

# Install a package
pip install numpy
```

Additional Tips:

- **Environment Names:** Choose descriptive names for your environments to easily identify their purpose.
- **Environment Variables:** Set environment variables to customize environment settings.
- **Shared Environments:** Create shared environments for common dependencies used across multiple projects.
- **Environment Management Tools:** Explore GUI-based tools like Anaconda Navigator for easier environment management.

By effectively using Python environments, you can streamline your development workflow, enhance project organization, and collaborate more efficiently with others.

Creating and Using Python Environments on Windows with venv

1. Ensure Python is Installed Make sure you have Python installed on your Windows machine. You can download it from the official Python website (https://www.python.org/).

2. Open a Command Prompt or PowerShell Right-click on the Start menu and select "Command Prompt" or "Windows PowerShell".

3. Create a Virtual Environment Use the `venv` module to create a new virtual environment:
Cmd /Powershell

```
python -m venv my_env
```

Replace `my_env` with the desired name for your environment. This will create a new directory named `my_env` with the necessary files for the virtual environment.

4. Activate the Virtual Environment To use the newly created environment, activate it:
Cmd /Powershell

```
my_env\Scripts\activate
```

You should see the environment name in parentheses before your prompt, indicating that it's active.

5. Install Packages Now, you can install packages specific to your project:
Cmd /Powershell

```
pip install numpy pandas matplotlib
```

These packages will be installed within the `my_env` directory, isolated from your system-wide Python installation.

6. Deactivate the Environment To exit the virtual environment:
Cmd /Powershell

```
deactivate
```

Example: Cmd /Powershell

```
# Create a new environment
python -m venv my_project

# Activate the environment
my_project\Scripts\activate

# Install a package
pip install flask

# Run a Python script
python my_script.py
```

Note:

- You can create multiple virtual environments for different projects.
- To reuse an environment for a new project, simply activate it and install the necessary packages.
- For more advanced environment management, consider using tools like `conda`.

By following these steps, you can effectively manage Python environments on Windows, ensuring that your projects have their own isolated dependencies and configurations.