**Title**

Reading Multiple Notes In a TXT File

**Snippet Code**

with open("FileName.txt", "r") as file:  
 # Read the lines from the file  
 lines = file.readlines()

# Iterate through the lines   
 for i in range(0, len(lines), \_\_number of lines each note has\_\_:   
 # Extract the attributes of the note from the lines  
 title = lines[i].strip()  
 text = lines[i + 1].strip()

# Add more attributes if note has more such as links and tags

# Print the note details  
 print(title)  
 print("Text:", text)  
 # Add more print statements if there are more details

**Title**

Save as csv file

**Text**

This snippet allows you to save as a csv file

**Snippet Code**

with open("notes1.csv", "w", newline='') as csvfile:

fieldnames = ['title','text','meta']

writer = csv.DictWriter(csvfile,fieldnames=fieldnames)

writer.writeheader()

for note in notes:

writer.writerow(note)

**Title**

Python List Comprehensions

**Text**

List comprehensions provide a concise way to create lists in Python. They consist of an expression followed by a for clause, and optionally, additional for or if clauses. List comprehensions are a powerful tool for creating lists by applying an expression to each item in an iterable.

**Snippet code**

# Using a list comprehension to create a list of squares  
squares = [x\*\*2 for x in range(10)]

# Using conditions in a list comprehension  
even\_squares = [x\*\*2 for x in range(10) if x % 2 == 0]

# List comprehension with if-else expression  
numbers = [1, 2, 3, 4, 5]  
transformed\_numbers = [x if x % 2 == 0 else x\*2 for x in numbers]

**Title**

Python's Zip Function

**Text**

The zip() function in Python is used to combine multiple iterables (like lists) element-wise into a single iterable.

**Snippet Code**

l1=[1,2,3]

l2 = ['a', 'b','c']

zipp = zip(l1,l2)

zlist = list(zipp)

print(zlist)

the console would return [(1,'a') , (2,'b'), (3,'c')]

#zip() #function

**Title**

Series and DataFrame

**Text**

Pandas has two main data structures, "Series" and "DataFrame." Series is a 1D array, and can include labels. DataFrame is a 2D array, and has a named + ordered column structure; each column can contain a different data type (e.g. string and boolean).

**Snippet Code**

series = pd.Series)array\_1D)

dataframe = pd.DataFrame(array\_2D)

**Title**

Pandas basics

**Text**

Pandas is a data analysis and manipulation tool in python. Pandas has three data structures:deries, dataframe, and index objects.Some of the functionalities include: Reindexing, axis oeprations (drop/add), filter, sorting and ranking, etc.

**Snippet Code**

import pandas as pd

mydataset = {  
 'cars': ["BMW", "Volvo", "Ford"],  
 'passings': [3, 7, 2]  
}

myvar = pd.DataFrame(mydataset)

print(myvar)

links:https://www.w3schools.com/python/pandas/pandas\_getting\_started.asp

Tags: #Pandas, #basics

**Title**

Error Handling in Python Using Try-Except Blocks

**Text**

Error handling is an essential aspect of programming to gracefully handle unexpected situations and prevent program crashes. Python provides the try-except block for handling exceptions.

**Snippet Code**

try:

# Code block that may raise an exception

result = 10 / 0 # ZeroDivisionError

except ZeroDivisionError:

# Code block to handle the exception

print("Error: Division by zero occurred.")

**Title**

Using Python's enumerate() Function for Iterating Over Lists

**Text**

Python's enumerate() function is a useful built-in function for iterating over elements in a list while also keeping track of the index of each element.

**Snippet Code**

# Example: Iterating over a list with enumerate()

fruits = ['apple', 'banana', 'cherry', 'date']

for index, fruit in enumerate(fruits):

print(f"Index {index}: {fruit}")

**Title**

Blackjack Ace Function

**Text**

# Manually selects what value Ace will be for dealer when drawn  
def dealer\_ace(ace\_drawn):  
 """  
 Used only for the dealer. Automatically determines whether Ace is equal to 1 or 11 based on Blackjack conventions.

Arguments:  
 ace\_drawn (string): the Ace the dealer drew  
   
 Returns:  
 ace\_drawn (string): the dealer's current Ace

**Snippet Code**

total = dealer\_draw(card\_deck)

if total >= 10:

ace\_drawn = 1

print("For this draw, Ace equals {}".format(ace\_drawn))

return ace\_drawn

else:

ace\_drawn = 11

print("For this draw, Ace equals {}".format(ace\_drawn))

return ace\_drawn

**Title 10**

Python Dictionary Comprehension

**Text**

Python Dictionary Comprehension Example

This creates a dictionary with numbers as keys and their squares as values.

**Snippet Code**

squared\_dict = {num: num\*num for num in range(1, 11)}

print(squared\_dict)  
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}

**Title**

String Formatting in Python

**Text**

String Formatting in Python Example

This demonstrates different methods for formatting strings in Python.

**Snippet Code**

name = "Alice"  
age = 30

# Using f-strings (formatted string literals)  
formatted\_string = f"Hello, my name is {name} and I am {age} years old."  
print(formatted\_string)

# Using the .format() method  
formatted\_string = "Hello, my name is {} and I am {} years old.".format(name, age)  
print(formatted\_string)

# Using %-formatting (old-style formatting)  
formatted\_string = "Hello, my name is %s and I am %d years old." % (name, age)

print(formatted\_string)

**Title**

Tkinter Window Rule

**Text**

Everything that has to do w the window object has to go in between these commands.

**Snippet Code**

window = Tk()

window.mainloop()

**Title**

Python Inheritance

**Text**

Inheritance allows us to define a class that inherits all the methods and properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived class.

The child class named "Student" will inherit the properties and methods from the parent class named "Person"

**Snippet Code**

class Person:  
 def \_\_init\_\_(self, fname, lname):  
 self.firstname = fname  
 self.lastname = lname

def printname(self):  
 print(self.firstname, self.lastname)

class Student(Person):  
 pass

**Title**

Python Polymorphism

**Text**

The word "Polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

For example, here we have 3 classes: Car, Boat, and Plane, and they all have a method called move()

**Snippet Code**

class Car:

def \_\_init\_\_(self, brand, model):

self.brand = brand

self.model = model

def move(self):

print("Drive!")

class Boat:

def \_\_init\_\_(self, brand, model):

self.brand = brand

self.model = model

def move(self):

print("Sail!")

class Plane:

def \_\_init\_\_(self, brand, model):

self.brand = brand

self.model = model

def move(self):

print("Fly!")

**Title**

Inheritance

**Text**

Inheritance allows us to define a class and inherit all the methods and properties of that class. For example, if I create a parent class, then all the attributes and methods get passed down to the child class.

**Snippet Code**

class Person:  
 def \_\_init\_\_(self, fname, lname):  
 self.firstname = fname  
 self.lastname = lname  
  
 def printname(self):  
 print(self.firstname, self.lastname)  
  
class Student(Person):  
 pass

x = Student("Mike", "Olsen")  
x.printname()

**Title**

Lambda

**Text**

Lambdas is a small anonymous function that can take any number of arguments, but it is constricted to only one expression and one expression only.

**Snippet Code**

This adds ten to the argument a

x = lambda a : a + 10  
print(x(5)

This multiplies a with argument b

x = lambda a, b : a \* b

print(x(5, 6))

Now this add the total result for a, b, and c

x = lambda a, b, c : a + b + c

print(x(5, 6, 2))

**Title 17**

Objects

**Text**

Objects can be a flexible way to store information or functions

**Title**

Dictionary

**Text**

Dictionaries can be very useful in storing information like this because they are easy to extend and can store multiple attributes

**Title**

Submit Method, creating a dictionary

**Snippet Code**

title = self.note\_title.get() # extract user input from title section

text = self.note\_text.get('1.0', 'end').strip('\n') # extract user input from text section

tag = self.note\_tags.get() # extract user input from tags section

link = self.note\_link.get() # extract user input from link section

meta = meta

note\_dict = {'title':title, 'text':text, 'link':link, 'tag':tag, 'meta':meta} # create the dictionary by creating key value pairs

make\_note = MakeNote(note\_dict) # pass the dictionary through to the creation of the MakeNote instance called make\_note

self.notes.append(make\_note) # append the instance of the class to the notes variable

self.notebook.append(make\_note) # append the instance of the class to the notebook variable

self.destroy() # destroy the window

return note\_dict # returns note\_dict variable which is the dictionary

**Title 20**

Methods in Python

**Text**

When you are coding and you are noticing that part of your code is constantly being repeated, this is a sign that you could need to use a method to simplify your code. Methods in python can be run whenever you call the methods, and you can pass data through them from the class in the form of objects.

**Title**

Structuring a repository

**Text**

To structure a Python repository effectively, organize it with folders for documentation, source code, tests, and examples. Utilize setup files like setup.py and setup.cfg to define installation instructions and metadata for the package. Include a requirements.txt file listing development dependencies, a README.rst file for project information, and a .gitignore file to specify which files Git should ignore. Additionally, configure a tox.ini file for running unit tests across multiple Python versions using tox, ensuring code compatibility and reliability.

**Title**

Using dictionaries

**Text**

Dictionaries in Python store data in key-value pairs, providing an ordered and changeable collection. Dictionaries maintain the order of insertion, ensuring consistency in iteration. Additionally, dictionaries do not allow duplicate keys, with duplicate key assignments resulting in overwriting existing values.

**Title**

Add Metadata

**Snippet Code**

# possibly to add metadata

self.metadata = metadata  
def add\_metadata (self, value)

self.metadata[note] = value # note would be in separate function

**Title**

Creating simple flask API

**Snippet Code**

from flask import Flask, jsonify

app = Flask(\_\_name\_\_)

@app.route('/')  
def hello\_world():  
 return 'Hello, World!'

@app.route('/api/data')  
def get\_data():  
 data = {'example\_data': [1, 2, 3, 4, 5]}  
 return jsonify(data)

if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(debug=True)

**Title**

Working with Named Tuples in Python

**Text**

Named tuples in Python provide a convenient way to define simple classes without the overhead of full class definitions. You can define a named tuple using the collections.namedtuple() function. You can access fields using dot notation or indexing. Named tuples are immutable. Named tuples improve code readability by providing named access to tuple elements.

**Title**

Classes

**Text**

When you make programs in a more complex way, use classes

**Title**

Class Inheritance in Python

**Text**

In Python, inheritance is a key concept in object-oriented programming that allows classes to inherit attributes and methods from other classes. This promotes code reuse and helps in building hierarchical relationships between classes.

**Snippet Code**

class Animal:  
 def sound(self):  
 pass

class Dog(Animal):  
 def sound(self):  
 return "Woof!"

class Cat(Animal):  
 def sound(self):  
 return "Meow!"

dog = Dog()  
print(dog.sound()) # Output: Woof!

cat = Cat()  
print(cat.sound())

**Title**

Python Decorators

**Text**

Decorators in Python are a powerful tool for modifying or extending the behavior of functions or methods without changing their actual implementation. They are functions that take another function as input and return a new function that usually extends or modifies the behavior of the original function. Decorators are commonly used for tasks such as logging, caching, access control, and more.

**Snippet Code**

def my\_decorator(func):  
 def wrapper():  
 print("Something is happening before the function is called.")  
 func()  
 print("Something is happening after the function is called.")  
 return wrapper

@my\_decorator  
def say\_hello():  
 print("Hello!")

say\_hello()

**Title**

Introduction to Pandas DataFrames

**Snippet Code**

# Pandas is a powerful library for data manipulation and analysis in Python.  
# It provides data structures like Series and DataFrame, which are efficient for working with structured data.

# Importing Pandas  
import pandas as pd

# Creating a DataFrame  
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],  
 'Age': [25, 30, 35, 40],  
 'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']}  
df = pd.DataFrame(data)  
print(df)

# Accessing columns  
print(df['Name']) # Accessing the 'Name' column  
print(df.Age) # Another way to access the 'Age' column

# Accessing rows  
print(df.iloc[0]) # Accessing the first row

# Adding a new column  
df['Gender'] = ['Female', 'Male', 'Male', 'Male']  
print(df)

# Filtering data  
print(df[df['Age'] > 30]) # Filtering rows where Age is greater than 30

**Title 30**

Polymorphism

**Text**

Polymorphism is one of the key pillars of OOP. It means "having different forms", which in Python translates to allowing for a single function to take on multiple forms depending on the object it's instantiated in. For example, with polymorphism, one operation will behave differently across different classes. Polymorphism is useful because it allows for a single interface to create different types of objects.

**Title**

Encapsulation

**Text**

Encapsulation is another key pillar of OOP. In involves containing the methods involved with data and the data itself within a singular unit. This is what classes are. Classes contain attributes and methods which are then applied to whatever objects are created using the classes. These objects have clear properties and uses.

**Title**

What is Object-Oriented Programming in Python?

**Text**

OOPs in python are a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming.

**Title**

Calling Parent class method

**Text**

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. Parent class methods can also be called within the overridden methods. This can generally be achieved by using Parent’s class methods can be called by using the Parent classname.method inside the overridden method.

**Title**

How to use datetime module

**Snippet Code**

import datetime as dt  
# renames the datetime module to: dt

now = dt.datetime.now()  
year = now.year  
month = now.month  
day\_of\_week = now.weekday()  
print(day\_of\_week)

date\_of\_birth = dt.datetime(year=2003, month=10, day=24, hour=4)  
print(date\_of\_birth)

**Title**

Back To The Beginning: OOP

**Text**

In OOP, there are 3 main parts to it. Encapsulation which is collecting data to put it together, Polymorphism which is when objects are able to do different things, and Inheritance when objects can have same features.

**Title**

Inheritance in OOP

**Text**

Inheritance in OOP is like taking material from the older category and putting it into the new one. In other words, your reusing a code and just adding some new things to it. This is beneficial in OOP because it keeps everything organized.

**Title**

Function to check if a list is empty

**Text**

This function checks if a list is empty and returns a Boolean value (True or False).

**Snippet Code**

def is\_list\_empty(lst):  
 """Check if the given list is empty and return a Boolean result."""  
 return len(lst) == 0

# Example Usage  
my\_list = []  
print(is\_list\_empty(my\_list)) # Output: True

**Title**

Exploring Nested Loops for Multi-dimensional Data Processing

**Text**

Nested loops are a powerful construct in Python that allow you to iterate over multi-dimensional data structures such as lists of lists or matrices. They enable you to traverse each element in a nested structure and perform operations efficiently. Nested loops are commonly used in tasks like matrix manipulation, image processing, and searching through multi-dimensional data.

**Title**

Data Cleaning with Python

**Snippet Code**

import pandas as pd

data = pd.read\_csv('your\_dataset.csv')

# Dropping duplicate rows  
data = data.drop\_duplicates()

# Handling missing values by filling them with mean  
data.fillna(data.mean(), inplace=True)

#Removing outliers   
from scipy import stats  
data = data[(np.abs(stats.zscore(data)) < 3).all(axis=1)]

#Saving cleaned data into a new CSV file  
data.to\_csv('cleaned\_data.csv', index=False)

**Title 40**

RegEx

**Text**

Regular Expressions are very useful for data analysis using python. To import you would type "import re", this basically allows you easy search for terms. It could be used for the notebook application to search which specific note in a huge list has the information sequence that you need.

**Title**

Syntax

**Text**

Typically, in coding you want to make sure everything is indented to the same level. You can do this by utilizing 4 spaces for all indentations instead of using tab which has a chance to mess things up. Also for function names they should be lower case.