ENGINEERING ACADEMY

Python Workshop

intercodex

Content

intercodex

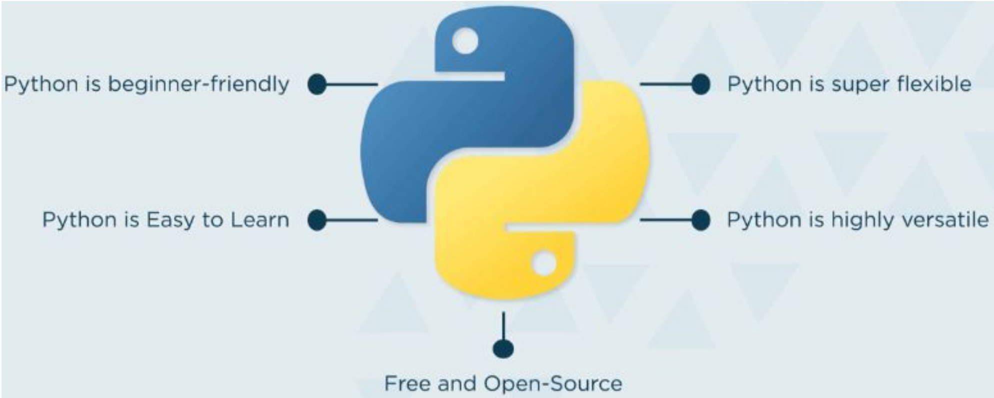
Introduction to Python

What is Python?

- High-level, interpreted language

Key Features

- Readability – perfect for beginners and experts
- Simplicity – takes very little to get started
- Large library collection – from core SE to Data Science, ML, etc.



Application Areas

- Data Science – data mining, exploration, preprocessing
- ML Engineering – Data preparation, unsupervised/supervised learning, computer vision, deep learning, NLP

Intuitive

- Designed for readability: `print("Hello, Python!")`
- Interpreted Language – highly flexible, no specific entry point

intercodex

Python Basics

Data Types

- Fundamentals include integers (int), floats (float), strings (str), and booleans (bool).
- Collections like lists, tuples, and dictionaries organize data.
- Example:

```
age = 25 (int)

names = ["Alice", "Bob"] (list)

person = {"name": "Mark", "age": 25, "address": {"street": "123 ABC
Boulevard", "city": "Mangaung", "postal code": 1234}} (dict)
```

Control Structures

- Make decisions with if-else, loop with for and while.

Python Basics Cont...

- Example:

```
if age >= 18:
    print("Adult")
else:
    print("Child")
```

Python Functions

- Reusable code blocks with def. Simplifies tasks, enhances readability.
- Example:

```
def greet(name: str) -> str:
    return f"Hi, {name}"
```

Python Functions & Libraries

Libraries extend core capabilities, offer enhanced tools for wide range of tasks
Examples:

- The requests library simplifies HTTP requests. Install via pip install requests

```
import prequests
def get_website_status(url: str):
    return requests.get(url).status_code
```

- pandas provides structures and functions for data analysis. Install with pip install pandas

```
import pandas
def load_csv(file_path: str) -> :
    return pandas.read_csv(file_path)
```

- matplotlib is used for creating visualizations. Install via pip install matplotlib

```
def plot_data(x, y):
    matplotlib.pyplot.plot(x, y)
    matplotlib.pyplot.show()
```

Virtual Environments

Virtual environments are isolated spaces for project-specific dependencies

- They ensure no conflicts can exist between project dependencies
- They Retain the relevant versions of dependencies for safeguarding against updates

Creating a virtual environment:

```
python -m venv env_name
```

Check libraries installed:

```
pip freeze
```

Activate virtual environment:

```
source env_name/bin/activate
```

Deactivate:

```
deactivate
```

intercodex

Managing Dependencies

- Managing dependencies keeps library versions consistent across different setups
- Normally done through the use of a requirements.txt file:
    - Lists all Python packages/libraries used in your project
    - Includes versions that are all compatible with each other
    - Ensures replication can be done
- Create a requirements.txt file:

1. Through pip freeze:

    ```
    pip freeze > requirements.txt
    ```

2. Through pipreqs:

    ```
    pipreqs .
    ```

3. Install packages from requirements.txt (remember to activate virtual environment):

    ```
    pip install -r requirements.txt
    ```

Data Preprocessing with Pandas

- Pandas is a data manipulation and analysis tool, especially for handling tabular data like CSV
- Can handle various formats like CSV, TXT, Excel, JSON and even database like SQL
- Example:

```
import pandas as pd
df = pd.read_csv("retail_sales_data.csv")
```

- Explore your data using head(), info(), and describe() to get structure, types, and statistics
- Example:

```
df.head()
df.info()
```

- Identify and fill missing data using methods like isna() and fillna(), crucial for data integrity
- Example:

```
df.isna().sum()
df.fillna(method="ffill")
```

intercodex

Data Preprocessing with Pandas

- Duplicates can skew analysis. Use drop_duplicates() to remove duplicate rows
- Example:

```
df = df.drop_duplicates()
```

- Transform data using operations like sorting, filtering, and aggregating
- Example:

```
df.sort_values(by="Sales", ascending=False)
df["Sales"].mean()
```

- Group data and perform calculations per group using groupby(). Essential for segment-wise analysis in retail
- Example:

```
total_sales_per_cat = df.groupby("Category").sum()
avg_sales_per_cat = df.groupby("Category")["Sales"].mean()
```

intercodex

Data Preprocessing with Pandas

- Merge or concatenate multiple DataFrames for comprehensive analysis. Methods like merge() and concat() are used for combining data
- Example:

```
df_merged = pd.concat([df1, df2])
```

```
df_merged = df1.merge(df2, on="ProductID")
```

intercodex

Scientific Computing with Numpy

- Numpy offers structures, particularly arrays, and tools for numerical operations.
- Numpy arrays are efficient, flexible. Basic operations include array creation, reshaping, and indexing.
- Example:

```
np_array = np.array([1, 2, 3, 4, 5])
```

- Reshape arrays for different data views, and perform slicing for data access.
- Example:

```
np_array.reshape(1, 5)
np_array[:3] (Slicing)
```

- Perform vectorized operations like addition, subtraction, and multiplication
- Example:

```
np_array * 2
```

intercodex

Scientific Computing with Numpy

- Numpy provides a wide range of statistical functions, such as mean, median, and standard deviation.
- Example:

```
np.mean(np_array)
```

- Use Numpy for advanced operations like matrix multiplication and determinants
- Example:

```
import numpy as np


def dot_multiplication(matrix_a, matrix_b):
    return np.dot(matrix_a, matrix_b)


matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])
```

Scientific Computing with Numpy

```
result = dot_multiplication(matrix_a, matrix_b)
print(f"Dot Product of matrix_a and matrix_b: \n{result}")
```

- Example:

```
def calculate_determinant(matrix):
      return np.linalg.det(matrix)


square_matrix = np.array([[2, 3], [1, 4]])


determinant = calculate_determinant(square_matrix)
print(f"Determinant of square_matrix: {determinant}")
```

intercodex

Scientific Computing with Numpy

Summary: Numpy is a powerful library for efficiently performing advanced computations.

1. Array Creation, Reshaping and Manipulation. Create & operate on arrays & matrices
2. Linear Algebra. Compute matrix dot & cross products, inverses, determinants, solve linear equations, eigenvalues, eigenvectors, matrix rank...
3. Type Conversions. Convert array data types (int, float, etc.), coerce arrays to other types
4. Logical Operations. Perform element-wise >, <, =, AND, OR, NOT…
5. Mathematical Functions. Basic arithmetic, trigonometric and hyperbolic functions, exponential and logarithmic functions, advanced functions like sine, cosine.
6. Statistical Operations. Compute mean, median, variance, standard deviation, min, max, sum, product, cumulative sum, product….
7. Random Module. Generate random numbers from various distributions
8. Input/Output. Read/write arrays to/from files in various formats, save and load…

intercodex

Data Visualization with matplotlib

- matplotlib enables diverse types of charts, catering to different data presentation needs
- Why Visualize? key for insights, making data more understandable.
- Example (line chart):

```
from matplotlib import pyplot as plt
import pandas as pd


data = pd.load_csv(“sales_data.csv”)
def plot_line_chart(data):
    plt.plot(data["Month"], data["Sales"])
    plt.title("Monthly Sales")
    plt.show()
```

Data Visualization with matplotlib Cont...

- Example (bar chart):

```
def plot_bar_chart(data):
    plt.bar(data["Month"], data["Sales"], label="Sales")
    plt.bar(data["Month"], data["Expenses"], label="Expenses",
alpha=0.7)
    plt.legend()
    plt.title("Sales vs Expenses")
    plt.show()
```

Data Visualization with matplotlib Cont…

- Example (pie chart):

```
def plot_pie_chart(data):
    plt.pie(data["Profit"], labels=data["Month"], autopct="%1.1f%%")
    plt.title("Profit Distribution"); plt.show()
```

- Example (scatter plot):

```
def plot_scatter_chart(data):
    plt.scatter(data["Sales"], data["Profit"])
    plt.xlabel("Sales")
    plt.ylabel("Profit")
    plt.title("Sales vs. Profit Analysis")
    plt.show()
```

intercodex

Object Oriented Programming

Classes are blueprints for creating objects, encapsulating data and functions together.

Example:

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price
    def name(self):
        return self.name
    def price(self):
        return self.price

milk = Product(“Milk”, 2.49)
```

intercodex

Object Oriented Programming

Example:

class Employee:

```
def __init__(self, name, id_number, department):
        self.name = name
        self.id_number = id_number
        self.department = department

    def display_details(self):
        # Instance method to display employee information
        return f"Employee: {self.name}, ID: {self.id_number}, Department: {self.department}"
```

intercodex

Object Oriented Programming

```
class Manager(Employee): # Manager is a subclass of Employee
    def __init__(self, name, id_number, department, managed_department):
        super().__init__(name, id_number, department)
        self.managed_department = managed_department

    def display_details(self): # Overriding the method to add managed
department
        basic_details = super().display_details()
        return f"{basic_details}, Managed Department:
{self.managed_department}"
def employee_summary(employee): # Polymorphism: Function works for any
Employee subclass
    print(employee.display_details())
```

intercodex

Class Relationships - Inheritance

- Inheritance is a mechanism where a new class inherits properties and behaviors (methods) from an existing class
- Types include Single, Multiple, Multilevel, Hierarchical, and Hybrid Inheritance
- Promotes code reusability, adds robustness, and helps in creating a hierarchical classification
- Example:

```
class Vehicle:
   def drive():
      pass

class Car(Vehicle):
   # Car inherits vehicle, therefore can drive
    pass
```

intercodex

Class Relationships - Composition and Aggregation

- Composition implies a 'part-of' relationship where the life cycle of the contained object depends on the container
- Aggregation implies a 'has-a' relationship where the contained object can exist independently
- Key difference: Composition: strong association, Aggregation: weak association
- Composition is used for more dependent relationships, whereas Aggregation is used where independence is desired
- Example:

```
class Engine:
    pass
class Car:
    def __init__(self):
        self.engine = Engine()  # Composition
```

intercodex

Class Relationships - Composition and Aggregation

Example:

```
class Team:
    def __init__(self, members):
        self.members = members  # Aggregation
class Employee:
    pass
# Employees can exist independently of a team
employee1 = Employee()
employee2 = Employee()
team = Team([employee1, employee2])
```

intercodex

OOP Best Practices

- Don"t Repeat Yourself (DRY)- Use functions, classes and patterns to avoid repeating logic
- Example:

```
class EmployeeReport:
    def __init__(self, employees):
        self.employees = employees


    def generate_full_time_report(self):
        report = "Full-Time Employees:\n"
        for emp in self.employees:
            if emp["type"] == "full-time":
                report += f"{emp["name"]} - {emp["position"]}\n"
        return report
```

OOP Best Practices

```
    def generate_part_time_report(self):
        report = "Part-Time Employees:\n"
        for emp in self.employees:
            if emp["type"] == "part-time":
                report += f"{emp["name"]} - {emp["position"]}\n"
        return report
```

intercodex

OOP Best Practices

```
class EmployeeReport:
    def __init__(self, employees):
        self.employees = employees
    def _generate_report(self, emp_type):
        report = f"{emp_type.title()} Employees:\n"
        for emp in self.employees:
            if emp["type"] == emp_type:
                report += f"{emp["name"]} - {emp["position"]}\n"
        return report
    def generate_full_time_report(self):
        return self._generate_report("full-time")
    def generate_part_time_report(self):
        return self._generate_report("part-time")
```

intercodex

OOP Best Practices

- SOLID Principles - A set of five design principles for good software engineering, making code more understandable, flexible, and maintainable.
    - S: Single Responsibility - A class should have only one job or responsibility
    - O: Open/Closed Principle - Entities (classes, modules, functions) should be open for extension but closed for modification
    - L: Liskov Substitution Principle - Objects of a superclass should be replaceable with objects of its subclasses without breaking the application
    - I: Interface Segregation Principle - No client should be forced to depend on methods it does not use. Create specific interfaces rather than one general-purpose interface
    - D: Dependency Inversion Principle - High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

intercodex

OOP Best Practices

- S: Single Responsibility - A class should have only one job or responsibility
- Example: (Bad)

```
class User:
    def __init__(self, name: str):
        self.name = name
    def get_user_data(self): # Retrieve user data from database
        pass
    def save_user_data(self, user_data): # Save user data to a database
        pass
    def email_user(self, content): # Send an email to the user
        pass
```

OOP Best Practices

```
class User:
    def __init__(self, name: str):
        self.name = name
class UserDataManager:
    @staticmethod
    def get_user_data(user: User): # Retrieve user data from database
        pass
    @staticmethod
    def save_user_data(user: User, user_data): # Save user data to a database
        pass
class EmailService:
    @staticmethod
    def email_user(user: User, content): # Send an email to the user
```

OOP Best Practices

- O: Open/Closed Principle - Entities should be open for extension but closed for modification
- Example: (Bad)

```
class ReportGenerator:
    def generate_report(self, report_type):
        if report_type == "PDF":
            # Generate PDF report
            pass
        elif report_type == "Word":
            # Generate Word report
            pass
```

intercodex

OOP Best Practices

- Example: (Good)

```
class ReportGenerator:
    def generate_report(self, report_formatter):
        report_formatter.format_report()
class PDFReportFormatter:
    def format_report(self):
        # Generate PDF report
        pass
class WordReportFormatter:
    def format_report(self):
        # Generate Word report
        pass
```

intercodex

OOP Best Practices

- L: Liskov Substitution Principle - Objects of a superclass should be replaceable with objects of its subclasses without breaking the application
- Example: (Bad)

```
class Bird:
    def fly(self):
        # Implement flying behavior

class Penguin(Bird):
    def fly(self):
        raise NotImplementedError("Penguins can't fly")
```

OOP Best Practices

- Example: (Good)

```
class Bird:
    pass


class FlyingBird(Bird):
    def fly(self):
        # Implement flying behavior


class Penguin(Bird):
    pass
```

OOP Best Practices

- I: Interface Segregation Principle - No client should be forced to depend on methods it does not use. Create specific interfaces rather than one general-purpose interface
- Example: (Bad)

```
class Worker:
    def work(self):
        pass
    def eat(self):
        pass
class Human(Worker):
    def work(self):
        pass # Working
    def eat(self):
        pass # Eating
```

intercodex

OOP Best Practices

- Example: (Good)

```
class Workable:
    def work(self):
        pass
class Eatable:
    def eat(self):
        pass
class Employee(Workable, Eatable):
    def work(self):
        pass # Working
    def eat(self):
        pass # Eating
class Child(Eatable):
```

intercodex

OOP Best Practices

- D: Dependency Inversion Principle - High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions. Example: (Bad)

```
class LightBulb:
    def turn_on(self):
        pass
    def turn_off(self):
        pass
class Switch:
    def __init__(self):
        self.bulb = LightBulb()
    def operate_switch(self):
        pass # Operate the switch
```

intercodex

OOP Best Practices

- Example: (Good)

```
class Switchable:
    def turn_on(self):
        pass
    def turn_off(self):
        pass
class LightBulb(Switchable):
    def turn_on(self):
        pass # Turn on the light
    def turn_off(self):
        pass # Turn off the light
```

intercodex

Design Patterns

- Set of software design solutions designed to solve common problems and limitations to building applications
- They represent best practices to design of often complex problems
- They also model the real world in a more concrete way
- Can be thought of as templates that can be applied to real-world programming scenarios
- Three groups: creational, structural and behavioural. 23 in total, but we'll only consider 15
- Example:

```
class Vehicle:
    def __init__(self, no_of_cyl: int, engine_size: int, fuel_type: category):
        # Set attributes
hilux = Vehicle(no_of_cyl = 6, engine_size = 5000, fuel_type = "Diesel")
# Now let's build a Tesla
???
```

intercodex

Design Patterns - Creational

- Provide mechanisms for object creation
- Enhances code flexibility, reuse by abstracting instantiation process
- This makes applications independent of how objects are created, composed and represented
- Increases simplicity - refer to above point
- Improves scalability, maintainability & reusability of objects
- Very important: client doesn't worry about how objects are created

intercodex

Design Patterns - Creational - Singleton

- Ensures a class has only one instance and provides a global point of access to it
- Subsequent attempts to create an instance return the existing instance
- Ensures consistency across the application
- Restricts object creation, offering controlled access
- Efficient use of resources, especially for connections and logging
- Ensures data consistency across the application
- Use for:
  - Configuration Files - Single instance to hold application settings
  - Database Connections - Managing database connections
  - Logging - Single logging utility throughout the application
  - Caution - Use judiciously, avoid overuse

Design Patterns - Creational - Singleton

- Example:

```
class SingletonMeta(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(SingletonMeta, cls).__call__(*args,
**kwargs)
        return cls._instances[cls]

class DatabaseConnection(metaclass=SingletonMeta):
```

intercodex

Design Patterns - Creational - Factory Method

- Provides interface for creating objects in superclass but allows subclasses to alter the type of objects that will be created
- Purpose is to delegate the instantiation logic to child classes
- Useful in scenarios where class instantiation may involve complex logic
- Encapsulates object creation, making code more flexible and reusable
- Structure & components:
  - Creator - An abstract class that declares the factory method
  - Concrete Creator - A subclass that implements or overrides the factory method
  - Product - An interface for the type of object the factory method creates
  - Concrete Product - A subclass that implements the Product interface

Design Patterns - Creational - Factory Method

- Easily introduce new products without changing existing code
- Reduces dependencies between application and concrete classes
- Single Responsibility: The creation logic is kept separate from the main business logic
- Use when a class wants its subclasses to specify the objects it creates
- Most important: used when class implementation needs to be decoupled from its usage
- Simplifies code by moving the creation logic to a single place

Design Patterns - Creational - Factory Method

```
from abc import ABC, abstractmethod
class Transport(ABC): #Creator Class - Declares the factory method.
    @abstractmethod
    def create_transport(self): # All the child classes must implement this
        pass
    def plan_delivery(self):
        transport = self.create_transport() # Common function
        print(f"Delivery planned with: {transport.delivery_method()}")
class Truck(Transport):
    def create_transport(self):
        return RoadTransport()
road_delivery = Truck()
road_delivery.plan_delivery()
```

intercodex

Design Patterns - Creational - Abstract Factory

- Allows the creation of families of related or dependent objects without specifying their concrete classes
- Abstract Factory pattern works around a super-factory that creates other factories
- Emphasizes on the approach of creating objects through interfaces and not through concrete classes
- Structure & components:
  - Abstract Factory - An interface with methods for creating abstract products
  - Concrete Factory - Implements the operations to create concrete products
  - Abstract Product - Declares a type of product object
  - Concrete Product - A product object of a family

intercodex

Design Patterns - Creational - Abstract Factory

- Clients use interfaces, not specific implementations
- Easier to exchange product families
- Ensures products are compatible within a family by ensuring consistent creation pattern
- Used when the system should be independent of the way its products are created, composed, and represented
- Use dwhen families of related products are designed to be used together
- Enhances modularity of a system and facilitates interchangeability of families

intercodex

Design Patterns - Creational - Abstract Factory

```
from abc import ABC, abstractmethod
# Abstract Factory Interface
class AnimalFactory(ABC):
    @abstractmethod
    def create_animal(self):
        pass
# Concrete Factories
class MammalFactory(AnimalFactory):
    def create_animal(self):
        return Mammal()
class BirdFactory(AnimalFactory):
    def create_animal(self):
        return Bird()
```

intercodex

Design Patterns - Creational - Abstract Factory

```
class ReptileFactory(AnimalFactory):
    def create_animal(self):
        return Reptile()
# Abstract Product Interface
class Animal(ABC):
    @abstractmethod
    def number_of_limbs(self):
        pass
    @abstractmethod
    def type_of_covering(self):
        pass
```

intercodex

Design Patterns - Creational - Abstract Factory

```python
# Concrete Products
class Mammal(Animal):
    def number_of_limbs(self):
        return 4
    def type_of_covering(self):
        return "Fur"
class Bird(Animal):
    def number_of_limbs(self):
        return 2  # Two wings, two legs
    def type_of_covering(self):
        return "Feathers"
```

intercodex

Design Patterns - Creational - Abstract Factory

```
class Reptile(Animal):
    def number_of_limbs(self):
        return 4

    def type_of_covering(self):
        return "Scales"
```

Design Patterns - Creational - Abstract Factory

```
# Usage
mammal_factory = MammalFactory()
mammal = mammal_factory.create_animal()
print(f"Mammal: Limbs - {mammal.number_of_limbs()}, Covering -
{mammal.type_of_covering()}")
bird_factory = BirdFactory()
bird = bird_factory.create_animal()
print(f"Bird: Limbs - {bird.number_of_limbs()}, Covering -
{bird.type_of_covering()}")
reptile_factory = ReptileFactory()
reptile = reptile_factory.create_animal()
print(f"Reptile: Limbs - {reptile.number_of_limbs()}, Covering -
{reptile.type_of_covering()}")
```

intercodex

Design Patterns - Creational - Builder

- Separates the construction of a complex object from its representation
- Allows the creation of a complex object step-by-step, and constructs different types or representations of an object using the same construction process
- Ideal for constructing complex objects with numerous fields and nested objects
- Used when an object needs to be created with many optional components or configurations
- Structure & components:
  - Builder - An abstract interface for creating parts of a complex object
  - Concrete Builder - Implements the Builder interface and provides an interface for retrieving the product
  - Director - Constructs an object using the Builder interface
  - Product - The complex object being built

Design Patterns - Creational - Builder

- Main advantage: construction process is isolated from main business logic
- Code for initialization is more readable, manageable and logical
- Important: allows constructing objects with various configurations from same building process
- Used when the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
- Example: you've been hired as a lead software engineer for McDonald's, and you have to build the control logic for an automated meal maker (staffless shop). The shop has screens that take user input and send it to your API in the following format:

```
OrderID: 123
Burger: chicken
Side: salad
Drink: water
Dessert: fruit salad
```

intercodex

Design Patterns - Creational - Builder

- Example:

```
class Meal:
    def __init__(self, burger, side, drink, dessert):
        self.burger = burger
        self.side = side
        self.drink = drink
        self.dessert = dessert

    def __str__(self):
        return (f"Meal with a {self.burger} burger, "
                f"{self.side} side, a {self.drink} drink, "
                f"and a {self.dessert} dessert.")
```

intercodex

Design Patterns - Creational - Builder

```python
class MealBuilder:
    def __init__(self):
        self.burger = 'beef'
        self.side = 'fries'
        self.drink = 'cola'
        self.dessert = 'apple pie'
    def set_burger(self, burger_type):
        self.burger = burger_type
        return self
    def set_side(self, side_type):
        self.side = side_type
        return self
```

intercodex

Design Patterns - Creational - Builder

```
def set_drink(self, drink_type):
        self.drink = drink_type
        return self
    def set_dessert(self, dessert_type):
        self.dessert = dessert_type
        return self
    def build(self):
        return Meal(self.burger, self.side, self.drink, self.dessert)
mb = MealBuilder()
meal =
(mb.set_burger('chicken').set_side('salad').set_drink('water').set_dessert('fr
uit salad').build())
print(meal)
```

Design Patterns - Creational - Prototype

- Lets you copy existing objects without making your code dependent on their classes
- Used to allow the creation of new objects by copying existing ones
- Provides a way to clone objects without coupling to their specific classes
- Ideal where object creation is costly, avoids duplication of code
- Involves implementing a cloning interface in the base class, typically with a clone method
- Useful when the number of classes in a system makes the class hierarchy too large
- Structure & components:
    - Prototype - An abstract class or interface that defines the clone method
    - Concrete Prototype - A subclass that implements the cloning method
    - Client - Creates new objects by asking a prototype to clone itself

intercodex

Design Patterns - Creational - Prototype

- Used when the classes to instantiate are specified at runtime
- Used when cloning an object is more desirable than creating it afresh due to performance considerations
- Used for avoiding creation of a factory hierarchy needed to create an object
- Can clone the object and reconfigure the clone for specific application
- Example: you are hired by a tech company, ThemeForest, as a software engineer for automating the generation of new website themes & templates for the company to sell
- You get an API call from the UX team with the template theme:

```
ThemeID: euwm-3984-fm43-g59v
Layout: Standard
ColorScheme: Blue & White
Font: Arial
```

- You are to clone this template and apply dark theme & dark gray color scheme

intercodex

Design Patterns - Creational - Prototype

- Example:

```
import copy

class WebsiteTemplate:

    def __init__(self, name, layout, color_scheme, font):

        self.name = name

        self.layout = layout

        self.color_scheme = color_scheme

        self.font = font

    def clone(self):

        # Create a deep copy of the current template

        return copy.deepcopy(self)
```

intercodex

Design Patterns - Creational - Prototype

```
    def customize(self, name, color_scheme=None, font=None):
        # Customize specific attributes of the template
        self.name = name
        if color_scheme:
            self.color_scheme = color_scheme
        if font:
            self.font = font
    def __str__(self):
        return (f"Template: {self.name}\n"
                f"  Layout: {self.layout}\n"
                f"  Color Scheme: {self.color_scheme}\n"
                f"  Font: {self.font}")
```

intercodex

Design Patterns - Creational - Prototype

```
# Base template
base_template = WebsiteTemplate("Base", "Standard", "Blue & White", "Arial")
# Create new theme by cloning and customizing base template
dark_theme = base_template.clone()
dark_theme.customize("Dark Theme", color_scheme="Dark Gray & Black")
minimalist_theme = base_template.clone()
minimalist_theme.customize("Minimalist Theme", color_scheme="Minimal White",
font="Helvetica")


print(base_template)
print(dark_theme)
print(minimalist_theme)
```

intercodex

Design Patterns - Structural

- Structural patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient (note: composition)
- Simplify complex structures by identifying simple ways to realize relationships between entities - this makes the overall system easier to understand and maintain
- Ensure that changes in one part of a system require minimal changes in other parts - loose coupling and strong encapsulation
- Allow for dynamic composition of behaviors
- Save time and effort in the design phase
- Promote interface compatibility

intercodex

Design Patterns - Structural - Adapter

- Allows objects with incompatible interfaces to collaborate
- Used when you want to use an existing class, and its interface does not match the one you need
- Involves separate adapter class that converts the (incompatible) interface of a class into another interface clients expect
- In other words, it's a bridge (middleware) between two (or more) incompatible interfaces
- Allows for communication between different systems, particularly when a legacy system is involved
- Can be used to convert data from one interface to another (like a translator)

intercodex

Design Patterns - Structural - Adapter

- Example:

```
class XMLPaymentSystem: # legacy system using XML
    def process_payment(self, xml):
        print(f"Processing payment: {xml}")
        return "<response><status>success</status></response>"

class APIPaymentSystem: # modern system using RESTful API
    def process_payment(self, json_data):
        print(f”Processing payment: {json_data}”)
        return {
          “status”: “success”
          “amount”: json_data[amount]
      }
```

intercodex

Design Patterns - Structural - Adapter

```
import json


import xml.etree.ElementTree as ET
class PaymentManager(APIPaymentSystem): # Adapter interface
    def __init__(self, legacy_system):
        self.legacy_system = legacy_system


    def process_payment(self, json_data):
        data = json.loads(json_data)
        xml_data = f"<payment><amount>{data['amount']}</amount></payment>"
        response_xml = self.legacy_system.process_payment(xml_data)
        response_et = ET.ElementTree(ET.fromstring(response_xml))
        return json.dumps({"status": response_et.find("./status").text})
```

Design Patterns - Structural - Composite

- Pattern for composing objects into tree structures to represent part-whole hierarchies
- Creates a tree structure of group objects and individual objects in a way that they can be treated uniformly
- Clients can treat both single objects and compositions of objects in the same way
- Structure and composition:
  - Composite objects - Define behavior for components having children. They store child components and implement child-related operations in the Component interface
  - Leaf objects - perform the actual tasks, while composite objects store child components. Leaf objects have no children
- Clients use the Component class interface to interact with objects in the composite structure
- You can introduce new kinds of components without changing the code that uses the components, as long as they work through the Component interface

Design Patterns - Structural - Composite

- Example: file system:
  - Composite (Directory): Represents a directory that can contain files and other directories

```
from abc import ABC, abstractmethod
from typing import List
class FileSystemItem(ABC): # Component: represents both files and directories
    @abstractmethod
    def display(self) -> None:
        pass
class File(FileSystemItem): # Leaf: represents file in the system
    def __init__(self, name):
        self.name = name
    def display(self):
        print(f"File: {self.name}")
```

intercodex

Design Patterns - Structural - Composite

```
class Directory(FileSystemItem): # Composite: directory, can have files &
other directories
    def __init__(self, name):
        self.name = name
        self.children = []
    def add(self, item: FileSystemItem):
        self.children.append(item)
    def remove(self, item: FileSystemItem):
        self.children.remove(item)
    def display(self):
        print(f"Directory: {self.name}")
        for child in self.children:
            child.display()
```

intercodex

Design Patterns - Structural - Composite

```
# Usage
root = Directory("root")
file1 = File("file1.txt")
file2 = File("file2.txt")
subdir = Directory("subdir")

root.add(file1)
root.add(subdir)
subdir.add(file2)

root.display()  # Display entire tree
```

intercodex

Design Patterns - Structural - Proxy

- A proxy is a placeholder for another object to control access to it
- Used to create a representative object that controls access to another object, which may be remote, expensive to create, or in need of securing
- Three types of proxies:
  - Remote Proxy - Represents an object in a different address space (like a network)
  - Virtual Proxy - Used for lazy initialization of a heavy object (for faster performance)
  - Protection Proxy - Controls access to an object based on access rights
- Proxies significantly improve application performance by being temporary representations of a more computationally expensive object, only loading the object when it's really needed
- Example: loading media files, accessing data from a remote database, controlling access to employee payslips

Design Patterns - Structural - Proxy

- Example:

```
import time


class HighResolutionImage:
    def __init__(self, image_file):
        self.image_file = image_file
        time.sleep(2)  # Simulate time-consuming operation
        print(f"Image loaded: {image_file}")

    def display(self):
        print(f"Displaying {self.image_file}")
```

Design Patterns - Structural - Proxy

```
class ImageProxy:
    def __init__(self, image_file):
        self.image_file = image_file
        self.image = None
    def display(self):
        if self.image is None:
            self.image = HighResolutionImage(self.image_file)
        self.image.display()

proxy_image = ImageProxy("sample.jpg")
proxy_image.display()  # This will load and display the image
```

intercodex

Design Patterns – Structural – Flyweight

- Pattern used to reduce the memory footprint of a large number of similar objects
- Useful when a program requires a huge number of objects that don't vary much in state
- Saves memory by sharing as much data as possible with similar objects
- Key properties:
  - Intrinsic State - shared state which is stored in the flyweight objects. Independent of the flyweight's context and is immutable. Flyweight class is created with intrinsic state and functions
  - Extrinsic State - context-specific state that is not shared and must be provided by the client code. Passed into flyweight class as arguments
  - Flyweight Factory - creates and manages flyweight objects. It ensures that flyweights are shared correctly. When the client requests a flyweight, the factory either returns an existing instance or creates a new one if it doesn't exist

intercodex

Design Patterns – Structural – Flyweight

- Example: you have been hired by a gaming company to create a chess game. You have to start by optimizing how pieces are created so that the game is computationally efficient

```
class GamePiece: # Flyweight class, represents type of piece
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def display_piece_info(self):
        return f"{self.color} {self.name}"
```

Design Patterns - Structural - Flyweight

```
class PiecePosition: # Context, represents specific game piece & position
    def __init__(self, x, y, game_piece):
        self.x = x
        self.y = y
        self.game_piece = game_piece

    def display(self):
        print(f"Piece {self.game_piece.display_piece_info()} at position
({self.x}, {self.y})")
```

intercodex

Design Patterns - Structural - Flyweight

```
class PieceFactory: # Flyweight Factory for managing game pieces
    _pieces = {}

    @staticmethod
    def get_piece(name, color):
        key = (name, color)
        if key not in PieceFactory._pieces:
            PieceFactory._pieces[key] = GamePiece(name, color)
        return PieceFactory._pieces[key]
```

Design Patterns – Structural – Flyweight

```
white_king = PieceFactory.get_piece("King", "White")
black_queen = PieceFactory.get_piece("Queen", "Black")

pieces = [
    PiecePosition(0, 0, white_king),
    PiecePosition(7, 7, black_queen),
    PiecePosition(1, 1, white_king)  # Reuses the White King GamePiece
]

for piece in pieces:
    piece.display()

# Can then add some logic for how pieces can move,
```

intercodex

Design Patterns – Structural – Decorator

- Pattern that allows behavior to be added to individual objects dynamically
- This doesn't affect the behaviour of other objects in the class
- Used to extend or alter the functionality of objects at runtime by wrapping them in an object of a decorator class
- Key properties:
  - Component - original object to which the new functionality is added. It defines the interface for objects that can have responsibilities added to them dynamically.
  - Decorator - wraps the component and contains the additional behavior. Implements same interface as component and adds its own behavior either before or after delegating the task to the component
  - Concrete Component - specific object to which additional tasks are added
  - Concrete Decorator - specific decorator that adds responsibilities to the component

intercodex

Design Patterns - Structural - Decorator

- Example:

```
class Text: # Component, original base class, renders text
    def __init__(self, content):
        self.content = content
    def render(self):
        return self.content


class Decorator(Text): # Decorator, Text interface, delegates 'render' to it
    def __init__(self, text_component):
        self.component = text_component
    def render(self):
        return self.component.render()
```

Design Patterns - Structural - Decorator

```
class BoldDecorator(Decorator): # Concrete Decorator
    def render(self):
        return f"<b>{super().render()}</b>"
class ItalicDecorator(Decorator): # Concrete Decorator
    def render(self):
        return f"<i>{super().render()}</i>"
# Usage
simple_text = Text("Hello, World!")
bold_text = BoldDecorator(simple_text)
italic_and_bold_text = ItalicDecorator(bold_text)
print("Simple Text:", simple_text.render())
print("Bold Text:", bold_text.render())
print("Italic and Bold Text:", italic_and_bold_text.render())
```

intercodex

Conclusion

- Python Fundamentals:
  - Introduction to Python's syntax and basic constructs
  - Understanding Python data types, variables, and basic operations
  - Control structures including if-else statements, loops (for and while)
- Functions and Modules:
  - Writing and using Python functions for modular and reusable code
  - Importing and utilizing modules and packages in Python
- Object-Oriented Programming:
  - Core principles of OOP: Encapsulation, Inheritance, and Polymorphism
  - Creating classes and objects in Python
  - Implementing inheritance for code reuse and hierarchy representation

intercodex

Conclusion

- Advanced OOP Concepts:
    - Composition and Aggregation: Building complex objects
    - Understanding and implementing the DRY (Don't Repeat Yourself) principle
- Design Patterns:
    - Introduction to common design patterns in software development
    - Creational patterns like Singleton and Factory Method for object creation
    - Structural patterns like Adapter and Decorator for efficient class and object composition

intercodex