Control Structures

Control structures and functions are fundamental concepts in programming that enable developers to manage the flow of a program and execute specific tasks or actions. Control structures are used to make decisions, repeat actions, and handle various scenarios, while functions allow for the modular organization of code into reusable units.

Exercises:

1. Write a Python program that checks if a given number is even or odd and prints the result. (use if statement)
2. Write a program that prints the multiplication table (from 1 to 10) for a given number (use loop)
3. Create a program that counts down from 10 to 1 and then prints " take off!". (use loop)
4. Create a function that verifies if a number is prime or not.
5. Generate a list of all the prime numbers less than 50 using list comprehension. Use the function you created previously.
6. Write a program that asks the user to guess a random number between 1 and 100 and provides hints if the guess is too high or too low until the user guesses correctly. (use loop and conditions)

Data Types and Data Structures

In computer science and programming, data types are classified into two main categories: primitive or basic data types and non-primitive or composite data types. Primitive or basic data types include Integers, Floating-Point Numbers, Boolean, and Sting. In Python, built-in non-primitive or composite data types are lists, tuples, sets, and dictionaries.

Integers (int) represent whole numbers (positive or negative) without a fractional part.

```
x = 5
```

Floating-Point Numbers (float) represent real numbers with a decimal point. In Python, the float type is used for floating-point numbers.

```
y = 3.14
```

Boolean (bool) represents two values, True and False, typically used for logical operations and conditions.

```
is_true = True
```

Strings are sequences of characters. In Python, strings are sometimes treated as a non-primitive data structure because they are iterable and can be manipulated like other data structures.

```
my_string = "Hello, World!"
```

Lists are ordered collections of elements that can be of different data types. Lists are dynamic and can grow or shrink as needed.

```
my_list = [1, 2, 3, 'apple']
```

Tuples are ordered collections of elements, like lists, but they are immutable, meaning their elements cannot be changed after creation.

```
my_tuple = (1, 2, 3, 'banana')
```

Sets are unordered collections of unique elements. They are used for membership testing and removing duplicates.

```
my_set = {1, 2, 3, 3, 4}
```

Dictionaries are key-value pairs that store and retrieve data with a unique key.

```
my_dict = {'name': 'John', 'age': 30}
```

Average Grades for ISEG Programs 2020

Exercises:

Consider the following average grades for various ISEG programs in 2020:

Average grade for "Economia": 164.0

Average grade for "Gestão": 167.5

Average grade for "MAEG" (Mathematics): 187.5

Average grade for "Management": 167.0

Average grade for "Economics": 162.5

Average grade for "Finance": 160.0

1. Create a list (listEconomics) with the average grades of economics programs. The list should include the average grades of "Economia" and "Economics."
2. Create a list (listManagement) with the average grades of management programs. The list should include the average grade of "Management," "Finance," and "Gestão."
3. Create a list (listMathematics) with the average grades of mathematics programs. The list should include the average grade of "MAEG."
4. What is the purpose of using the "extend" method? Create a new list with the previous lists (listEconomics, listManagement, listMathematics) using "extend" and call it "listAll1."
5. What is the purpose of using the "append" method? Create a new list with the previous lists (listEconomics, listManagement, listMathematics) using "append" and call it "listAll2."
6. How many elements does "listAll2" have?
7. How many elements does "listAll1" have?
8. What is the average grade if all the courses have the same weight?

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm or style that uses objects to organize and structure code. It is based on the concept of "objects," which can be thought of as self-contained units that contain both data (attributes) and the methods (functions or procedures) that operate on that data. OOP is designed to model real-world entities and their interactions in software development.

Key principles of object-oriented programming include Classes, Objects, Encapsulation, Inheritance, and Polymorphism.

Classes are blueprints or templates for creating objects. They define the structure (attributes) and behavior (methods) that objects of a particular type will have.

Objects are instances of classes. They represent specific, individual entities and can interact with other objects by exchanging messages (invoking methods).

Encapsulation is the practice of bundling data (attributes) and the methods that operate on that data within a single unit (an object or class). This helps hide the internal details of how an object works, providing data protection and abstraction.

Inheritance allows developers to create new classes based on existing classes. The new class inherits the attributes and methods of the parent class (superclass) and can extend or override them to add or modify functionality.

Polymorphism means that different objects can respond to the same method or message in a way that is appropriate for their specific class. This allows for more flexible and generic code.

OOP promotes modularity, reusability, and ease of maintenance in software development. It is widely used in various programming languages, including Java, C++, Python, and PHP, and it is a fundamental concept in modern software engineering.

Exercises:

1. Define a class Person with attributes name and age. Create two Person objects and print their details.
2. Create a base class Shape with methods area and perimeter. Create subclasses Circle and Rectangle that are inherited from the Shape class and calculate their respective areas and perimeters.
3. Create a BankAccount class with attributes balance and methods deposit and withdraw. Ensure that the withdraw method does not allow the balance to go negative.
4. Define a class Animal with a method make sound. Create subclasses Dog, Cat, and Cow that override the make sound method to produce different sounds.
5. Create a Library class that contains a list of Book objects. Each Book should have attributes like title and author. Implement methods to add and remove books from the library.
6. Modify the BankAccount class from exercise 3 to handle exceptions when attempting to withdraw more money than the balance. You can use try and except to catch the exception.
7. Create a Library class that contains a list of Book objects. Each Book should have attributes like title and author. Implement methods to add and remove books from the library.

Programming Exercises (Solutions)

Control Structures

1. Write a Python program that checks if a given number is even or odd and prints the result. (use if statement)

```
number = int(input("Enter a number: "))
if number % 2 == 0:
    print(f"{number} is even.")
else:
    print(f"{number} is odd.")
```

2. Write a program that prints the multiplication table (from 1 to 10) for a given number.

```
number = int(input("Enter a number: "))
for i in range(1, 11):
      print(f"{number} x {i} = {number * i}")
```

3. Create a program that counts down from 10 to 1 and then prints " take off!". (use loop)

```
countdown = 10
while countdown > 0:
    print(countdown)
    countdown -= 1
print("take off!")
```

4. Create a function that verifies if a number is prime or not.

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

5. Generate a list of all the prime numbers less than 50 using list comprehension. Use the function you created previously.

```
prime_numbers = [x for x in range(2, 50) if is_prime(x)]
print(prime_numbers)
```

6. Write a program that asks the user to guess a random number between 1 and 100 and provides hints if the guess is too high or too low until the user guesses correctly. (use loop and conditions)

```
import random
secret_number = random.randint(1, 100)
guess = 0
while guess != secret_number:
    guess = int(input("Guess the number: "))
    if guess < secret_number:
        print("Too low! Try again.")
    elif guess > secret_number:
        print("Too high! Try again.")
    else:
        print(f"Congratulations! You guessed the secret number: 
{secret_number}")
```

Data Types and Data Structures

Consider the following average grades for various ISEG programs in 2020:

Average grade for "Economia": 164.0

Average grade for "Gestão": 167.5

Average grade for "MAEG" (Mathematics): 187.5

Average grade for "Management": 167.0

Average grade for "Economics": 162.5

Average grade for "Finance": 160.0

1. Create a list (listEconomics) with the average grades of economics programs. The list should include the average grades of "Economia" and "Economics."

```
listEconomics=[economia, economics]

economia=190.0

print(listEconomics)
```

2. Create a list (listManagement) with the average grades of management programs. The list should include the average grade of "Management," "Finance," and "Gestão."

```
listManagment=[gestao,  management,finance]

print(listManagment)
```

3. Create a list (listMathematics) with the average grades of mathematics programs. The list should include the average grade of "MAEG."

```
listMathmatics=[maeg]

print(listMathmatics)
```

4. What is the purpose of using the "extend" method? Create a new list with the previous lists (listEconomics, listManagement, listMathematics) using "extend" and call it "listAll1."

```
listAll1 = []

listAll1.extend(listEconomics)

listAll1.extend(listManagement)

listAll1.extend(listMathematics)
```

5. What is the purpose of using the "append" method? Create a new list with the previous lists (listEconomics, listManagement, listMathematics) using "append" and call it "listAll2."

   Solution: The "append" method is used to add an element to the end of a list.

```
listAll2 = listEconomics + listManagement + listMathematics
```

6. How many elements does "listAll2" have?

```
len(listAll2)
```

7. How many elements does "listAll1" have?

```
len(listAll1)
```

8. What is the average grade if all the courses have the same weight?

```
average_grade_all_courses = sum(listAll2) / len(listAll2)
```

Object-Oriented Programming:

1. Define a class Person with attributes name and age. Create two Person objects and print their details.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person1 = Person("Alice", 30)
person2 = Person("Bob", 25)
print(f"{person1.name} is {person1.age} years old.")
print(f"{person2.name} is {person2.age} years old.")
```

2. Create a base class Shape with methods area and perimeter. Create subclasses Circle and Rectangle that inherit from the Shape class and calculate their respective areas and perimeters.

```
import math
class Shape:
    def area(self):
        pass

    def perimeter(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius**2

    def perimeter(self):
        return 2 * math.pi * self.radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

circle = Circle(5)
rectangle = Rectangle(4, 6)
print("Circle Area:", circle.area())
print("Circle Perimeter:", circle.perimeter())
print("Rectangle Area:", rectangle.area())
print("Rectangle Perimeter:", rectangle.perimeter())
```

3. Create a BankAccount class with attributes balance and methods deposit and withdraw. Ensure that the withdraw method does not allow the balance to go negative.

```
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount

    def withdraw(self, amount):
        if amount > 0 and self.balance >= amount:
            self.balance -= amount
        else:
            print("Insufficient funds!")

account = BankAccount(1000)
account.deposit(500)
account.withdraw(300)
account.withdraw(1200)  # Should print "Insufficient funds!"
print("Balance:", account.balance)
```

4. Define a class Animal with a method make_sound. Create subclasses Dog, Cat, and Cow that override the make_sound method to produce different sounds.

```
class Animal:
    def make_sound(self):
        pass

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

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

class Cow(Animal):
    def make_sound(self):
        return "Moo!"
dog = Dog()
cat = Cat()
cow = Cow()
print(dog.make_sound())
print(cat.make_sound())
print(cow.make_sound())
```

5. Create a Library class that contains a list of Book objects. Each Book should have attributes like title and author. Implement methods to add and remove books from the library.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def remove_book(self, book):
        if book in self.books:
            self.books.remove(book)

library = Library()
book1 = Book("Python Programming", "John Doe")
book2 = Book("Data Science Essentials", "Jane Smith")
library.add_book(book1)
library.add_book(book2)
print("Books in the library:")

for book in library.books:
    print(f"{book.title} by {book.author}")

library.remove_book(book1)
print("After removing a book:")

for book in library.books:
    print(f"{book.title} by {book.author}")
```

6. Modify the BankAccount class from exercise 3 to handle exceptions when attempting to withdraw more money than the balance. You can use try and except to catch the exception.

```
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount

    def withdraw(self, amount):
        try:
            if amount > 0:
                if self.balance >= amount:
                    self.balance -= amount
                else:
                    raise ValueError("Insufficient funds!")
            else:
                raise ValueError("Invalid withdrawal amount.")
        except ValueError as e:
            print(e)

account = BankAccount(1000)
account.deposit(500)
account.withdraw(300)
account.withdraw(1200)  # Should print "Insufficient funds!"
print("Balance:", account.balance)
```

7. Create a Library class that contains a list of Book objects. Each Book should have attributes like title and author. Implement methods to add and remove books from the library.

```
class Book:
  def __init__(self, title, author):
    self.title = title
    self.author = author
class Library:
  def __init__(self):
    self.books = []
  def add_book(self, book):
    self.books.append(book)
  def remove_book(self, book):
    if book in self.books:
      self.books.remove(book)
```

```
library = Library()
book1 = Book("Python Programming", "John Doe")
book2 = Book("Data Science Essentials", "Jane Smith")

library.add_book(book1)
library.add_book(book2)

print("Books in the library:")
for book in library.books:
  print(f"{book.title} by {book.author}")

library.remove_book(book1)
print("After removing a book:")
for book in library.books:
  print(f"{book.title} by {book.author}")
```