Python Foundations Notes

1. Variables

Variables are containers for storing data values. In Python, you don't need to declare the data type; it is assigned automatically based on the value.

Rules for variable names:
- Must start with a letter or underscore (_)
- Cannot start with a number
- Case-sensitive (age and Age are different)

Example:

```
x = 10        # integer
y = 3.14      # float
name = "Alice"  # string
is_active = True  # boolean

print(x, y, name, is_active)
```

Output:
10 3.14 Alice True

2. Conditionals

Conditionals allow decision making using if, elif, and else.

Syntax:

```
if condition:
   # code
elif another_condition:
   # code
else:
   # code
```

Example:

```
marks = 75
```

```
if marks >= 90:
   print("Excellent")
elif marks >= 50:
   print("Pass")
else:
   print("Fail")
```

Output:
Pass

3. Loops

Loops help repeat actions.
- For loop: Iterates over a sequence.
- While loop: Repeats until condition is false.

Example:

```
for i in range(1, 4):
   print("Hello")

count = 0
while count < 3:
   print("Hi")
   count += 1
```

Output:
Hello
Hello
Hello
Hi
Hi
Hi

4. Functions

Functions are reusable blocks of code that improve readability and reduce repetition.

Example:

```
def add(a, b):
   return a + b

print(add(5, 3))
```

Output:
8

5. Data Structures

List

Ordered, mutable, allows duplicates.

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "mango"
print(fruits)
```

Output:
['apple', 'mango', 'cherry']

Tuple

Ordered, immutable, allows duplicates.

```
colors = ("red", "green", "blue")
print(colors[0])
```

Output:
red

Set

Unordered, no duplicates.

```
nums = {1, 2, 2, 3}
print(nums)
```

Output:
{1, 2, 3}

Dictionary

Stores data as key-value pairs.

```
student = {"name": "Alice", "age": 20}
print(student["age"])
```

Output:
20

6. File Handling

Python provides open() function with modes:
- 'r' → read
- 'w' → write (overwrite)
- 'a' → append

Example:

```
with open("data.txt", "w") as f:
   f.write("Hello, File Handling!")

with open("data.txt", "r") as f:
   print(f.read())
```

Output:
Hello, File Handling!

7. Modules

Modules are Python files with reusable functions.

Example:

```
import math

print(math.sqrt(25))
print(math.factorial(5))
```

Output:
5.0
120

8. Object-Oriented Programming (OOPs)

There are 4 OOP concepts. They are:

1. Polymorphism
2. Inheritance
3. Encapsulation
4. Abstraction

Object-Oriented Programming (OOP) in Python is a programming paradigm that structures programs around "objects" rather than functions and logic. It aims to model real-world entities and their interactions, leading to more organized, reusable, and maintainable code.

Key Concepts of OOP in Python:

Classes: Blueprints or templates for creating objects. They define attributes (data) and methods (functions) that objects of that class will possess.

Objects (Instances): Individual instances created from a class. Each object has its own unique set of instance attributes.

Encapsulation: The bundling of data (attributes) and methods that operate on the data within a single unit (the class). It also involves restricting direct access to some of an object's components, promoting data integrity.

Example program:

```
class Student:

def _init_(self, name, marks):

    self.__name = name      # private variable

    self.__marks = marks    # private variable

  # Getter

  def get_marks(self):

    return self.__marks

  # Setter

  def set_marks(self, marks):
```

```
        if 0 <= marks <= 100:
            self.__marks = marks
        else:
            print("Invalid marks!")
# Usage
s = Student("Gayathri", 85)
print("Marks (before update):", s.get_marks())
s.set_marks(92)
print("Marks (after update):", s.get_marks())
s.set_marks(150)  # Invalid case
```

output:

```
marks (before update): 85
marks (after update): 92
Invalid marks!
```

Polymorphism: The ability of objects of different classes to respond to the same method call in a way that is specific to their own class. This often involves method overriding (redefining a method in a child class) or method overloading (defining multiple methods with the same name but different parameters). However, Python doesn't support true method overloading in the same way as some other languages.

Example program:

```
class Dog:
    def sound(self):
        return "Bark"
class Cat:
    def sound(self):
        return "Meow"
# Polymorphism in action
for animal in (Dog(), Cat()):
```

```
print(animal.sound())
```

output:

bark

meow

Inheritance: A mechanism where a new class (child/derived class) can inherit attributes and methods from an existing class (parent/base class). This promotes code reuse and establishes a hierarchical relationship.

Example program:

```
# Parent class

class Animal:

    def speak(self):

        return "I am an animal"

# Child class (inherits from Animal)

class Dog(Animal):

    def speak(self):

        return "I am a Dog, I Bark!"

# Usage

a = Animal()

d = Dog()

print(a.speak())

print(d.speak())
```

Output:

I am an animal

I am a Dog, I Bark !

Abstraction: Hiding complex implementation details and showing only the essential features of an object. This is often achieved through abstract classes and methods, which define an interface without providing a complete implementation.

Example program:

```
# Abstract Class

class Shape(ABC):

   @abstractmethod

   def area(self):

      pass

# Subclass must implement abstract method

class Circle(Shape):

   def _init_(self, radius):

      self.radius = radiusdef area(self):return 3.14 * self.radius * self.radius

# Usage

c = Circle(5)

print("Area of Circle:",c.area())
```

output:

Area of a circle: 78.5

9. Data Types in Python

Python has built-in data types to represent different kinds of values. Data types define what kind of operations can be performed on the values.

Numeric Types

- int: Whole numbers (positive, negative, or zero).
- float: Numbers with decimal points.
- complex: Numbers in the form a + bj.

```
x = 10       # int
y = 3.14     # float
z = 2 + 3j   # complex

print(x, type(x))
print(y, type(y))
print(z, type(z))
```

Output:
10 <class 'int'>
3.14 <class 'float'>
(2+3j) <class 'complex'>

String (str)

A string is a sequence of characters enclosed in single, double, or triple quotes.

```
text = "Python is fun"
print(text.upper())
print(text[0:6])
```

Output:
PYTHON IS FUN
Python

Boolean (bool)

Represents True or False values, often used in conditions.

```
a = True
b = False
print(10 > 5)
print(5 == 2)
```

Output:
True
False

Sequence Types

List

Ordered, mutable, allows duplicates.

```
fruits = ["apple", "banana", "cherry"]
fruits.append("mango")
print(fruits)
```

Output:
['apple', 'banana', 'cherry', 'mango']

Tuple

Ordered, immutable, allows duplicates.

```
numbers = (1, 2, 3, 4)
print(numbers[1])
```

Output:
2

Range

Represents a sequence of numbers, often used in loops.

```
r = range(1, 6)
print(list(r))
```

Output:
[1, 2, 3, 4, 5]

Set Types

Set

Unordered, unique elements only (no duplicates).

```
nums = {1, 2, 2, 3}
print(nums)
```

Output:
{1, 2, 3}

Frozenset

Immutable set (cannot be changed).

```
f = frozenset([1, 2, 3, 2])
print(f)
```

Output:
frozenset({1, 2, 3})

Mapping Type - Dictionary

Stores key-value pairs. Keys must be unique.

```
student = {"name": "Alice", "age": 21, "course": "Python"}
print(student["name"])
```

Output:
Alice

None Type

Represents the absence of a value.

```
x = None
print(x)
print(type(x))
```

Output:
None
<class 'NoneType'>

✅ Summary of Python Data Types

- Numeric: int, float, complex
- String: str
- Boolean: bool
- Sequence: list, tuple, range
- Set Types: set, frozenset
- Mapping: dict
- None Type: NoneType