INTERNPE INTERNSHIP/TRAINING PROGRAM
Python 1 (NOTES CONTENT)

Introduction to Python

Python is a popular programming language known for its simplicity and versatility. It was created by Guido van Rossum and was first released in 1991. Python is designed with a focus on readability and ease of use, making it a great choice for beginners and experienced developers alike.

Why Python?

- **Easy to Read and Write:** Python's syntax is straightforward and resembles English language, making it easier to learn and write code quickly.
- **Versatility:** Python can be used for a wide range of applications, including web development, data analysis, artificial intelligence, scientific computing, and more.
- **Vast Community and Libraries:** Python has a huge community of developers, which means there's a wealth of resources, libraries, and frameworks available to help you solve various programming challenges.
- **Cross-Platform Compatibility:** Python is available on multiple platforms (Windows, macOS, Linux), so your code can be used on different operating systems without major modifications.
- **Interactive Mode:** Python has an interactive mode that allows you to test your code line by line, which is very helpful for learning and debugging.

Python's Philosophy - The Zen of Python (PEP 20)
The Zen of Python is a collection of guiding principles for writing computer programs in Python. It's a set of aphorisms that highlight the values and philosophies behind Python's design. You can access it by typing `import this` in your Python interpreter. Some key principles include:

- **Readability Counts:** Code is read more often than it's written. Write code that is easy to understand and maintain.
- **Simple is Better than Complex:** Strive for simplicity and clarity in your code rather than overcomplicating things.

- **Sparse is Better than Dense:** Avoid cramming too much code into a single line. Use whitespace and structure to make your code more readable.
- **Explicit is Better than Implicit:** Make your code's intentions clear and obvious rather than relying on hidden behaviours.

Setting Up Python Environment
Before you start writing Python code, you need to set up your development environment:

1. **Install Python:** Download the latest version of Python from the official website (https://www.python.org/downloads/). Follow the installation instructions for your operating system.

2. **Integrated Development Environment (IDE):** You can use an IDE like PyCharm, Visual Studio Code, or IDLE (Python's built-in IDE) to write and run your Python code. IDEs provide features like code completion, debugging, and project management.

3. **Text Editor:** Alternatively, you can use a plain text editor like Notepad (Windows) or nano (Linux/macOS) for writing Python code. However, this method lacks some features of dedicated IDEs.

4. **Running Python Code:** After writing your code, you can run it using the terminal or command prompt. Navigate to the directory where your Python file is located and use the command `python filename.py` to execute your code.

Now you're ready to start your journey into Python programming! You can experiment with basic code, explore different libraries, and gradually build your understanding of the language's concepts. Remember that practice is key to mastering any programming language.

Basic Syntax and Data Types

When you start programming in Python, you need to understand the basic rules and structure of the language. This section covers the fundamental elements like variables, data types, and basic operations.

Variables and Assignments
- **Variables:** Variables are like containers that hold data. They allow you to store and manipulate values in your code.

- **Assignment:** You can assign a value to a variable using the `=` symbol. For example, `x = 5` assigns the value 5 to the variable `x`.

Data Types: int, float, str, bool
- **int:** Stands for integer. Represents whole numbers, like 5, -10, 100.
- **float:** Stands for floating-point. Represents decimal numbers, like 3.14, -0.5.
- **str:** Stands for string. Represents text and is enclosed in single or double quotes, like 'hello', "Python".
- **bool:** Stands for boolean. Represents either `True` or `False`.

Basic Operators
Python supports a variety of operators for performing operations on data.

- **Arithmetic Operators:** Used for mathematical calculations.
  - `+` Addition: Adds two values.
  - `-` Subtraction: Subtracts right operand from left.
  - `*` Multiplication: Multiplies two values.
  - `/` Division: Divides left operand by right.
  - `%` Modulus: Returns the remainder after division.

- **Comparison Operators:** Compare two values and return a boolean result.
  - `==` Equal to: Checks if two values are equal.
  - `!=` Not equal to: Checks if two values are not equal.
  - `<` Less than: Checks if left value is less than right.
  - `>` Greater than: Checks if left value is greater than right.
  - `<=` Less than or equal to: Checks if left value is less than or equal to right.
  - `>=` Greater than or equal to: Checks if left value is greater than or equal to right.

- **Logical Operators:** Used to combine conditions.
  - `and` Logical AND: Returns `True` if both conditions are `True`.
  - `or` Logical OR: Returns `True` if at least one condition is `True`.
  - `not` Logical NOT: Returns the opposite of the condition's value.

Examples:
```python
x = 5
y = 2
name = "Alice"
is_valid = True

# Arithmetic
sum = x + y
difference = x - y
```

```
product = x * y
quotient = x / y
remainder = x % y

# Comparison
is_equal = x == y
is_greater = x > y

# Logical
both_conditions = is_valid and is_equal
either_condition = is_valid or is_greater
negated_condition = not is_valid
```

Understanding these basic concepts will lay the foundation for writing more complex programs in Python. As you continue learning, you'll explore more advanced features and techniques to build powerful applications.

Control Structures

Control structures are essential for directing the flow of your program. They enable you to make decisions, repeat actions, and create more dynamic and interactive code. Let's delve into the key control structures: conditional statements and loops.

Conditional Statements (if, elif, else)
Conditional statements allow you to execute specific code blocks based on certain conditions.

- **if Statement:** It checks a condition, and if it's true, the indented code block beneath it is executed.
```python
age = 18
if age >= 18:
    print("You're an adult!")
```

- **elif Statement:** Stands for "else if." It's used when you have multiple conditions to check.
```python
temperature = 25
if temperature > 30:
    print("It's hot!")
elif temperature > 20:
    print("It's warm.")
else:
```

```
    print("It's cool.")
```

- **else Statement:** This code block is executed if none of the previous conditions are true.
```python
grade = 85
if grade >= 90:
    print("A")
elif grade >= 80:
    print("B")
else:
    print("C or below")
```

****Loops (for, while)****
Loops allow you to repeat a set of instructions multiple times.

- ****for Loop:**** Iterates over a sequence (like a list, string, or range) and executes the indented code block for each item.
```python
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

- **while Loop:** Continues to execute a code block as long as a certain condition remains true.
```python
count = 0
while count < 5:
    print("Count:", count)
    count += 1
```

****Break and Continue Statements****
- **break Statement:** Terminates the loop prematurely when a specific condition is met.
```python
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 3:
        break
    print(num)
```

- **continue Statement:** Skips the remaining code in the current iteration and moves to the next iteration.
```python
numbers = [1, 2, 3, 4, 5]
```

```
for num in numbers:
    if num == 3:
        continue
    print(num)
```

Understanding control structures empowers you to make your programs more dynamic and responsive. They let you tailor your code's behavior to different scenarios and automate repetitive tasks. As you progress, you'll use these structures to build more complex and interactive applications.

Collections

Collections in Python are used to store and organize multiple values. They come in various forms, such as lists, tuples, dictionaries, and sets. These structures allow you to manage and manipulate groups of data efficiently.

Lists
A list is an ordered collection of items. Each item can be of any data type, and they are enclosed in square brackets `[]`.

```python
fruits = ["apple", "banana", "cherry"]
```

Tuples
Tuples are similar to lists, but they are immutable, meaning their elements cannot be changed after creation. Tuples are enclosed in parentheses `()`.

```python
coordinates = (3, 4)
```

Dictionaries
Dictionaries are collections of key-value pairs. Each value is associated with a unique key. Dictionaries are enclosed in curly braces `{}`.

```python
person = {
    "name": "Alice",
```

```
    "age": 30,
    "city": "New York"
}
```

Sets
Sets are unordered collections of unique elements. They are useful for storing distinct values and performing operations like union and intersection. Sets are enclosed in curly braces `{}`.

```python
unique_numbers = {1, 2, 3, 4, 5}
```

Accessing and Manipulating Collections

- **Accessing Elements:**
  - Lists and tuples: Elements are accessed using indices (0-based).
    ```python
    second_fruit = fruits[1]  # Accesses "banana"
    ```
  - Dictionaries: Values are accessed using keys.
    ```python
    person_age = person["age"]  # Accesses 30
    ```

- **Modifying Elements:**
  - Lists and tuples: Elements can be changed (for lists) or recreated (for tuples).
    ```python
    fruits[0] = "orange"  # Changes first element to "orange"
    ```
  - Dictionaries: Values associated with keys can be updated.
    ```python
    person["age"] = 31  # Updates age to 31
    ```

- **Adding and Removing Elements:**
  - Lists: Use `append()` to add an item, and `remove()` or `pop()` to remove items.
    ```python
    fruits.append("grape")  # Adds "grape" to the end
    fruits.remove("banana")  # Removes "banana"
    ```
  - Dictionaries: Use assignment to add or update items, and `del` to remove items.
    ```python
    person["gender"] = "female"  # Adds new key-value pair
    del person["city"]  # Removes "city" key-value pair

Collections provide powerful ways to organize, store, and manipulate data in your Python programs. They enable you to work with groups of related information and perform various
    ```

operations efficiently. As you become more comfortable with collections, you'll find them indispensable in many programming scenarios.

Functions

Functions are blocks of code that perform a specific task. They allow you to break down your code into smaller, reusable pieces, making your programs more organized and manageable. Functions play a crucial role in modular programming.

Defining Functions
To define a function, you give it a name and specify the code it should execute. You can also define parameters (inputs) that the function accepts.

```python
def greet(name):
    print(f"Hello, {name}!")
```

Calling Functions
Once you've defined a function, you can call it by using its name followed by parentheses. You can pass arguments (values) to the function's parameters.

```python
greet("Alice")  # Outputs: Hello, Alice!
```

Parameters and Return Values
Functions can take parameters, which are variables used as placeholders for values you'll provide when you call the function.

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

You can also have functions that don't return any value (implicitly return `None`), or you can use the `return` statement to explicitly return a value.

Lambda Functions
Lambda functions, also known as anonymous functions, are short, one-line functions that can be defined using the `lambda` keyword.

```python
multiply = lambda x, y: x * y
result = multiply(3, 4)  # Result is 12
```

****Built-in Functions****
Python comes with a set of built-in functions that are ready to use. These functions cover a wide range of operations, from mathematical calculations to working with strings.

```python
length = len("Python")  # Returns the length of the string: 6
maximum = max(4, 7, 2)  # Returns the maximum value: 7
minimum = min(4, 7, 2)  # Returns the minimum value: 2
```

****Advantages of Functions****
1. **Reusability:** You can use a function in multiple parts of your code, reducing redundancy.
2. **Modularity:** Functions help organize code into logical units, making it easier to understand and maintain.
3. **Abstraction:** Functions allow you to hide complex implementation details and provide a high-level interface.
4. **Debugging:** Smaller functions are easier to test and debug than large, monolithic code blocks.

Functions are the building blocks of programming. By using functions effectively, you can create more efficient, readable, and maintainable code.

Modules and Packages

In Python, modules and packages are tools that help you organize and structure your code. They allow you to break your code into manageable files and folders, making it easier to manage larger projects and collaborate with others.

****Modules****
A module is a single file containing Python code that defines functions, classes, and variables. You can use the contents of a module in other programs by importing it.

****Creating Modules:****
1. Create a new `.py` file with the desired module name, like `my_module.py`.
2. Write your functions, classes, or variables in the file.

Using Modules:
1. Import the module using `import`.
2. Access its contents using dot notation.

```python
# my_module.py
def greet(name):
    print(f"Hello, {name}!")

# main.py
import my_module
my_module.greet("Alice")  # Outputs: Hello, Alice!
```

Packages
A package is a collection of modules organized in a directory hierarchy. Packages allow you to group related modules together.

Creating Packages:
1. Create a directory for your package (folder).
2. Inside the package directory, create `.py` files as your modules.

Using Packages:
1. Import modules from packages using dot notation.

```python
# my_package/
# ├── __init__.py
# ├── module1.py
# └── module2.py

# main.py
from my_package import module1
module1.function_name()
```

Third-Party Packages
Apart from built-in modules and your own packages, you can also use third-party packages created by the Python community. These packages offer pre-written functionalities for various tasks.

Installing Third-Party Packages:
You can use the `pip` tool (Python's package manager) to install third-party packages.

```bash
pip install package_name
```

Using Third-Party Packages:
Once installed, you can import and use the package's modules in your code.

```python
import package_name
package_name.function_name()
```

Advantages of Modules and Packages
- **Organization:** Dividing code into modules and packages makes it easier to manage and navigate.
- **Reusability:** Modules and packages can be reused in multiple projects, saving time and effort.
- **Collaboration:** Large projects can be worked on collaboratively by assigning modules or packages to different team members.
- **Encapsulation:** Code in different modules/packages can have clear boundaries, preventing interference between parts of the codebase.

By utilising modules and packages, you can create well-structured, modular, and maintainable code, whether it's for your own projects or for sharing with the wider Python community.