Python Basics Notes:

1. Introduction to Python

Python is a high-level, interpreted, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant indentation. It supports multiple programming paradigms, including object-oriented, imperative, functional, and procedural styles.
Why Python?

- **Simplicity & Readability:** Python's syntax is simple and intuitive, making it easy to learn and read, resembling natural language.
- **Versatility:** Used in web development (Django, Flask), data science (NumPy, Pandas, Scikit-learn), artificial intelligence, machine learning, automation, scripting, desktop applications, and more.
- **Large Community & Ecosystem:** A vast number of libraries, frameworks, and a supportive community are available.
- **Portability:** Python code can run on various platforms (Windows, macOS, Linux) without modification.
- **Interpreted Language:** Code is executed line by line, simplifying debugging.

2. Installation

If you don't have Python installed, you can download it from the official website: python.org. It's recommended to install the latest stable version.

3. Basic Syntax and Concepts

3.1 Comments

Comments are used to explain code and are ignored by the Python interpreter.

```
# This is a single-line comment

'''
This is a multi-line comment
using triple single quotes.
'''

"""
This is also a multi-line comment
using triple double quotes.
"""
```

3.2 Variables

Variables are used to store data values. Python is dynamically typed, meaning you don't need to declare the type of a variable.

```
name = "Alice"   # String
age = 30         # Integer
height = 1.75    # Float
is_student = True # Boolean

print(name)
print(age)
print(height)
print(is_student)

# You can reassign variables with different types
age = "thirty"
print(age)
```

3.3 Data Types

Python has several built-in data types:

- **Numeric:** int, float, complex
- **Text:** str
- **Sequence:** list, tuple, range
- **Mapping:** dict
- **Set:** set, frozenset
- **Boolean:** bool (True, False)

```
# Numeric
integer_num = 10
float_num = 10.5
complex_num = 3 + 4j

# String
greeting = "Hello, Python!"

# List (ordered, mutable, allows duplicates)
my_list = [1, "apple", 3.14, True]

# Tuple (ordered, immutable, allows duplicates)
my_tuple = (10, "banana", 20)

# Dictionary (unordered, mutable, key-value pairs, keys must be
unique)
my_dict = {"name": "Bob", "age": 25, "city": "London"}

# Set (unordered, mutable, unique elements)
```

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

# Boolean
is_active = True
```

3.4 Operators

Arithmetic Operators

```
a = 10
b = 3

print(f"Addition: {a + b}")       # 13
print(f"Subtraction: {a - b}")    # 7
print(f"Multiplication: {a * b}") # 30
print(f"Division: {a / b}")       # 3.333... (float division)
print(f"Floor Division: {a // b}")# 3 (integer division)
print(f"Modulus: {a % b}")        # 1 (remainder)
print(f"Exponentiation: {a ** b}")# 1000 (10 to the power of 3)
```

Comparison Operators

Return True or False.

```
x = 5
y = 10

print(f"Equal to: {x == y}")         # False
print(f"Not equal to: {x != y}")     # True
print(f"Greater than: {x > y}")      # False
print(f"Less than: {x < y}")         # True
print(f"Greater than or equal to: {x >= y}") # False
print(f"Less than or equal to: {x <= y}")   # True
```

Logical Operators

and, or, not.

```
p = True
q = False

print(f"AND: {p and q}") # False
print(f"OR: {p or q}")   # True
print(f"NOT: {not p}")   # False
```

Assignment Operators

```
num = 5
num += 3 # num = num + 3
print(f"num after +=: {num}") # 8

num -= 2 # num = num - 2
print(f"num after -=: {num}") # 6

# Other assignment operators: *=, /=, //=, %=, **=
```

4. Control Flow

4.1 if, elif, else Statements

Used for conditional execution. Indentation defines code blocks.

```
score = 85

if score >= 90:
    print("Excellent!")
elif score >= 70:
    print("Good!")
else:
    print("Keep practicing.")

# Nested if-else
age = 18
has_id = True
if age >= 18:
    if has_id:
        print("You can enter.")
    else:
        print("Please show your ID.")
else:
    print("You are too young.")
```

4.2 for Loops

Iterate over a sequence (list, tuple, string, range, etc.).

```
# Iterate over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)

# Iterate using range()
```

```
for i in range(5): # 0, 1, 2, 3, 4
    print(f"Number: {i}")

# Iterate with index and value using enumerate()
for index, item in enumerate(fruits):
    print(f"Index {index}: {item}")
```

4.3 while Loops

Repeats a block of code as long as a condition is true.

```
count = 0
while count < 5:
    print(f"Count: {count}")
    count += 1 # Important to update the condition to avoid infinite
loop

# Using break and continue
i = 0
while i < 10:
    if i == 3:
        i += 1
        continue # Skip current iteration if i is 3
    if i == 7:
        break    # Exit loop if i is 7
    print(f"Looping: {i}")
    i += 1
```

5. Functions

Functions are reusable blocks of code that perform a specific task.

```
# Define a simple function
def greet(name):
    """
    This function greets the person passed in as a parameter.
    """
    print(f"Hello, {name}!")

# Call the function
greet("Alice")
greet("Bob")

# Function with return value
def add(x, y):
    """
    This function takes two numbers and returns their sum.
```

```
    """
    return x + y

result = add(5, 3)
print(f"Sum: {result}") # Output: 8

# Function with default parameters
def power(base, exponent=2):
    return base ** exponent

print(f"2 squared: {power(2)}")      # 4
print(f"2 cubed: {power(2, 3)}")      # 8

# Function with arbitrary arguments (*args)
def sum_all(*args):
    total = 0
    for num in args:
        total += num
    return total

print(f"Sum of 1, 2, 3: {sum_all(1, 2, 3)}") # 6

# Function with arbitrary keyword arguments (**kwargs)
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Carol", age=40, city="Paris")
```

6. Data Structures (Detailed)

6.1 Lists

Ordered, mutable (changeable), allows duplicate members.

```
my_list = [10, 20, 30, 40, 50]

# Accessing elements
print(f"First element: {my_list[0]}")     # 10
print(f"Last element: {my_list[-1]}")    # 50

# Slicing
print(f"Slice (index 1 to 3): {my_list[1:4]}") # [20, 30, 40]

# Modifying elements
my_list[0] = 100
print(f"Modified list: {my_list}") # [100, 20, 30, 40, 50]
```

```
# Adding elements
my_list.append(60)       # Add to end
my_list.insert(1, 15)    # Insert at specific index
print(f"After append and insert: {my_list}")

# Removing elements
my_list.remove(30)      # Remove by value
my_list.pop(0)          # Remove by index (returns removed item)
del my_list[2]          # Delete by index
print(f"After removals: {my_list}")

# Length
print(f"Length of list: {len(my_list)}")
```

6.2 Tuples

Ordered, immutable (unchangeable), allows duplicate members.

```
my_tuple = (1, 2, 3, "hello")

# Accessing elements (same as lists)
print(f"First element: {my_tuple[0]}") # 1

# Slicing (same as lists)
print(f"Slice: {my_tuple[1:3]}") # (2, 3)

# Tuples are immutable - cannot change elements
# my_tuple[0] = 100 # This would raise an error

# Unpacking a tuple
a, b, c, d = my_tuple
print(f"Unpacked: a={a}, b={b}, c={c}, d={d}")
```

6.3 Dictionaries

Unordered, mutable, key-value pairs, keys must be unique.

```
my_dict = {"name": "David", "age": 30, "city": "Berlin"}

# Accessing values
print(f"Name: {my_dict['name']}")
print(f"Age: {my_dict.get('age')}") # Safer: returns None if key not
found

# Adding/modifying key-value pairs
my_dict['email'] = 'david@example.com' # Add new
my_dict['age'] = 31                   # Modify existing
```

```
print(f"Updated dictionary: {my_dict}")

# Removing key-value pairs
my_dict.pop('city')             # Remove by key (returns value)
del my_dict['age']              # Delete by key
print(f"After removals: {my_dict}")

# Iterating through a dictionary
for key in my_dict:
    print(f"Key: {key}")

for value in my_dict.values():
    print(f"Value: {value}")

for key, value in my_dict.items():
    print(f"Key: {key}, Value: {value}")
```

6.4 Sets

Unordered, mutable, contains only unique elements.

```
my_set = {1, 2, 3, 2, 4}
print(f"Original set: {my_set}") # {1, 2, 3, 4} (duplicates removed)

# Adding elements
my_set.add(5)
my_set.update([6, 7])
print(f"After adding: {my_set}")

# Removing elements
my_set.remove(2) # Raises error if element not found
my_set.discard(10) # Does not raise error if element not found
print(f"After removing: {my_set}")

# Set operations
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}

print(f"Union: {set_a.union(set_b)}")             # {1, 2, 3, 4, 5, 6}
print(f"Intersection: {set_a.intersection(set_b)}") # {3, 4}
print(f"Difference (A - B): {set_a.difference(set_b)}") # {1, 2}
print(f"Symmetric Difference: {set_a.symmetric_difference(set_b)}") #
{1, 2, 5, 6}
```

7. String Manipulation

Strings are immutable sequences of characters.

```
my_string = "  Hello, Python Programming!  "

# Length
print(f"Length: {len(my_string)}") # 30

# Access characters
print(f"First character: {my_string[2]}")  # H (due to leading spaces)
print(f"Last character: {my_string[-1]}")  # (space)

# Slicing
print(f"Slice: {my_string[2:7]}") # Hello

# Stripping whitespace
print(f"Stripped: '{my_string.strip()}'") # 'Hello, Python
Programming!'

# Changing case
print(f"Uppercase: {my_string.upper()}")
print(f"Lowercase: {my_string.lower()}")

# Replacing substrings
print(f"Replace 'Python' with 'Java': {my_string.replace('Python',
'Java')}")

# Splitting strings
words = my_string.strip().split(' ')
print(f"Split into words: {words}") # ['Hello,', 'Python',
'Programming!']

# Joining strings
joined_string = "-".join(["alpha", "beta", "gamma"])
print(f"Joined string: {joined_string}") # alpha-beta-gamma

# String formatting (f-strings - preferred since Python 3.6)
name = "Eve"
age = 28
print(f"My name is {name} and I am {age} years old.")
```

8. File I/O

Reading from and writing to files.

```
# Writing to a file
# 'w' mode (write): creates file if not exists, truncates if exists
with open('my_file.txt', 'w') as file:
    file.write("This is the first line.\n")
```

```
    file.write("This is the second line.\n")
print("Content written to 'my_file.txt'.")

# Reading from a file
# 'r' mode (read): file must exist
with open('my_file.txt', 'r') as file:
    content = file.read()
    print("\nContent of 'my_file.txt':")
    print(content)

# Reading line by line
with open('my_file.txt', 'r') as file:
    print("\nReading line by line:")
    for line in file:
        print(line.strip()) # .strip() removes newline characters

# Appending to a file
# 'a' mode (append): adds content to the end of the file
with open('my_file.txt', 'a') as file:
    file.write("This is an appended line.\n")
print("\nContent appended to 'my_file.txt'.")

# Read again to see appended content
with open('my_file.txt', 'r') as file:
    print("\nContent after appending:")
    print(file.read())

# Clean up (optional)
import os
os.remove('my_file.txt')
print("\nCleaned up 'my_file.txt'.")
```

9. Error Handling (try, except, finally)

Used to handle runtime errors gracefully.

```
try:
    # Code that might raise an error
    result = 10 / 0
    print(result)
except ZeroDivisionError:
    # Code to execute if ZeroDivisionError occurs
    print("Error: Cannot divide by zero!")
except TypeError:
    # Code to execute if TypeError occurs
    print("Error: Type mismatch!")
except Exception as e:
```

```
    # Catch any other exception
    print(f"An unexpected error occurred: {e}")
else:
    # Code to execute if no error occurs in the try block
    print("Operation successful!")
finally:
    # Code that always executes, regardless of whether an error
occurred
    print("Execution complete.")

# Example with user input
while True:
    try:
        age = int(input("Enter your age: "))
        print(f"Your age is: {age}")
        break # Exit loop if input is valid
    except ValueError:
        print("Invalid input. Please enter a number.")
```

10. Modules and Packages

- **Module:** A file containing Python code (e.g., my_module.py).
- **Package:** A directory containing multiple modules and a special __init__.py file (which can be empty).

```
# Example: Using built-in modules
import math
print(f"PI: {math.pi}")
print(f"Square root of 16: {math.sqrt(16)}")

import random
print(f"Random integer between 1 and 10: {random.randint(1, 10)}")

# You can import specific functions/classes
from datetime import date
today = date.today()
print(f"Today's date: {today}")

# You can give aliases to imports
import numpy as np # Standard for NumPy
```