Pythan Basics Notes for ProgSD Exam

Contents

Chapter 1 Python Basics

1.1 Python Syntax

- Python uses indentation to define blocks of code.
- Comments start with `#`.
- Variable assignment is straightforward, and variables do not need explicit declaration.
- Common data types include:
  - `int`: Integer values.
  - `float`: Decimal numbers.
  - `str`: Text or string data.
  - `bool`: Boolean values (`True`, `False`).
  - `list`: Ordered, mutable sequence of items.
  - `tuple`: Ordered, immutable sequence of items.
  - `dict`: Key-value pairs.
  - `set`: Unordered, unique collection of items.
  - `NoneType`: Represents the absence of a value (`None`).

Example:

```
# Integer: 10
a = 10

# Float: 3.14
b = 3.14

# String: "Hello"
c = "Hello"

# Boolean: True
d = True

# List: [1, 2, 3]
e = [1, 2, 3]

# Tuple: (1, 2, 3)
f = (1, 2, 3)

# Dictionary: {’key’: ’value’}
g = {’key’: ’value’}

# Set: {1, 2, 3}
h = {1, 2, 3}

# NoneType: None
i = None
```

1.2 Variables and Data Types

- Variables store data that can be manipulated and reused.
- Common arithmetic operators include:
  - `+`: Addition.
  - `-`: Subtraction.
  - `*`: Multiplication.
  - `/`: Division (returns a float).
  - `//`: Floor division (discards the fractional part).
  - `%`: Modulus (remainder of division).
  - `**`: Exponentiation.
- Logical operators:
  - `and`: Returns `True` if both operands are `True`.
  - `or`: Returns `True` if at least one operand is `True`.
  - `not`: Negates a boolean value.

Example: Arithmetic Operators

```
# Addition
a = 10 + 5 # 15

# Subtraction
b = 10 - 5 # 5

# Multiplication
c = 10 * 5 # 50

# Division
d = 10 / 3 # 3.3333333333333335

# Floor Division
e = 10 // 3 # 3

# Modulus
f = 10 % 3 # 1

# Exponentiation
g = 2 ** 3 # 8
```

Example: Logical Operators

```
# Logical AND
result_and = (10 > 5) and (5 < 3) # False

# Logical OR
result_or = (10 > 5) or (5 < 3) # True

# Logical NOT
result_not = not (10 > 5) # False
```

1.3 Storing and Manipulating Values

- Variables can store data of various types, including numbers, strings, lists, tuples, dictionaries, and sets.
- Operations on variables depend on their data type and include arithmetic, concatenation, indexing, and more.

Example: Integer Operations

```
# Integer variable
a = 10

# Addition
b = a + 5 # 15

# Multiplication
c = a * 2 # 20

# Exponentiation
d = a ** 3 # 1000
```

Example: Float Operations

```
# Float variable
x = 3.14

# Division
y = x / 2 # 1.57

# Multiplication
z = x * 3 # 9.42
```

Example: String Operations

```
# String variable
s = "Hello"

# Concatenation
greeting = s + " World" # "Hello World"

# Repetition
repeated = s * 3 # "HelloHelloHello"

# Slicing
substring = s[1:4] # "ell"
```

Example: List Operations

```
# List variable
numbers = [1, 2, 3, 4]

# Append
numbers.append(5) # [1, 2, 3, 4, 5]

# Indexing
first_element = numbers[0] # 1

# Slicing
subset = numbers[1:3] # [2, 3]

# Concatenation
combined = numbers + [6, 7] # [1, 2, 3, 4, 5, 6, 7]
```

Example: Tuple Operations

```
# Tuple variable
coordinates = (10, 20, 30)

# Indexing
x_coord = coordinates[0] # 10

# Slicing
subset = coordinates[1:] # (20, 30)

# Concatenation
new_coords = coordinates + (40, 50) # (10, 20, 30, 40, 50)
```

Example: Dictionary Operations

```
# Dictionary variable
student = {"name": "Alice", "age": 25, "major": "Math"}

# Access value by key
name = student["name"] # "Alice"

# Add a new key-value pair
student["grade"] = "A" # {"name": "Alice", "age": 25,
    "major": "Math", "grade": "A"}

# Modify an existing value
student["age"] = 26 # {"name": "Alice", "age": 26, "major":
    "Math", "grade": "A"}
```

Example: Set Operations

```
# Set variable
unique_numbers = {1, 2, 3, 4}

# Add an element
unique_numbers.add(5) # {1, 2, 3, 4, 5}

# Union
other_numbers = {4, 5, 6}
union_set = unique_numbers.union(other_numbers) # {1, 2, 3,
    4, 5, 6}

# Intersection
intersection_set =
    unique_numbers.intersection(other_numbers) # {4, 5}
```

Example: Boolean Operations

```
# Boolean variable
flag = True

# Logical NOT
not_flag = not flag # False

# Logical AND
result = flag and False # False

# Logical OR
result_or = flag or False # True
```

Example: NoneType

```
# NoneType variable
x = None

# Checking if value is None
is_none = (x is None) # True
```

1.4 Working with Strings

- Strings in Python are immutable sequences of characters.
- Common operations include concatenation, slicing, and formatting.
- Useful string methods include `lower()`, `upper()`, `title()`, `strip()`, `split()`, and `len()`.

Example: Concatenation

```
# String concatenation
s1 = "Hello"
s2 = "World"
result = s1 + " " + s2 # "Hello World"
```

Example: Slicing

```
# String slicing
text = "Python Basics"
substring1 = text[0:6] # "Python"
substring2 = text[-6:] # "Basics"
```

Example: Formatting

```
# String formatting
name = "Alice"
age = 25
formatted = f"My name is {name} and I am {age} years old."
# "My name is Alice and I am 25 years old."
```

Example: Converting to Lowercase, Uppercase, and Title Case

```
# String methods
s = "hElLo WoRlD"

lowercase = s.lower() # "hello world"
uppercase = s.upper() # "HELLO WORLD"
titlecase = s.title() # "Hello World"
```

Example: Stripping Whitespace

```
# Removing leading and trailing whitespace
s = "  Python Basics  "
stripped = s.strip() # "Python Basics"
```

Example: Splitting a String

```
# Splitting a string into a list
s = "apple,banana,cherry"
split_list = s.split(",") # ["apple", "banana", "cherry"]
```

Example: String Length

```python
# Getting the length of a string
s = "Hello World"
length = len(s) # 11
```

Example: Finding Substrings

```python
# Finding a substring
s = "Hello World"
index = s.find("World") # 6
not_found = s.find("Python") # -1 (not found)
```

Example: Replacing Substrings

```python
# Replacing substrings
s = "Hello World"
replaced = s.replace("World", "Python") # "Hello Python"
```

1.5 Decision Making

- Python supports conditional branching with `if`, `elif`, and `else`.
- Logical operators like `and`, `or`, and `not` can combine conditions for more complex decisions.

Example: Basic `if` Statement

```
# Basic if statement
x = 10
if x > 5:
    result = "x is greater than 5" # "x is greater than 5"
```

Example: `if-else` Statement

```
# If-else statement
x = 3
if x > 5:
    result = "x is greater than 5"
else:
    result = "x is not greater than 5" # "x is not greater
        than 5"
```

Example: `if-elif-else` Statement

```
# If-elif-else statement
x = 5
if x > 5:
    result = "x is greater than 5"
elif x == 5:
    result = "x is equal to 5" # "x is equal to 5"
else:
    result = "x is less than 5"
```

Example: Logical Operators

```python
# Using logical operators
x = 10
y = 5

# Logical AND
if x > 5 and y < 10:
    result = "Both conditions are True" # "Both conditions
        are True"

# Logical OR
if x < 5 or y < 10:
    result = "At least one condition is True" # "At least
        one condition is True"

# Logical NOT
if not (x < 5):
    result = "x is not less than 5" # "x is not less than 5"
```

Example: Combining Logical Operators and `if-elif-else`

```python
# Combining logical operators with if-elif-else
x = 8
y = 3

if x > 5 and y > 5:
    result = "Both x and y are greater than 5"
elif x > 5 or y > 5:
    result = "At least one of x or y is greater than 5" #
        "At least one of x or y is greater than 5"
else:
    result = "Neither x nor y is greater than 5"
```

1.6 Repetition

- Python supports loops for repeating code:
  - `for` loops iterate over sequences like lists, tuples, or ranges.
  - `while` loops execute as long as a condition is `True`.
  - Nested loops allow iteration over multiple levels.

Example: Basic `for` Loop

```
# Iterating over a range
for i in range(5):
    result = i # 0, 1, 2, 3, 4
```

Example: `for` Loop with a List

```
# Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    result = fruit # "apple", "banana", "cherry"
```

Example: `while` Loop

```
# While loop with a condition
x = 0
while x < 5:
    x += 1 # 1, 2, 3, 4, 5
```

Example: Nested Loops

```
# Nested for loops
for i in range(2): # Outer loop
    for j in range(3): # Inner loop
        result = (i, j)
        # (0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)
```

Example: Nested `for` and `while` Loops

```
# Nested for and while loop
for i in range(3): # Outer loop
    x = 0
    while x < 2: # Inner loop
        result = (i, x)
        # (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
        x += 1
```

1.7 Functions

- Functions in Python encapsulate reusable blocks of code.
- Functions can:
  - Be defined using `def`.
  - Take parameters or have default arguments.
  - Return values or perform actions without returning.

Example: Defining a Function Without Parameters

```
# Function without parameters
def greet():
    result = "Hello, World!" # "Hello, World!"
```

Example: Defining a Function With Parameters

```
# Function with parameters
def add(a, b):
    result = a + b # Sum of a and b
```

Example: Function With Return Value

```
# Function that returns a value
def multiply(a, b):
    return a * b

result = multiply(2, 3) # 6
```

Example: Function Without Return Value

```
# Function without a return statement
def print_message(message):
    output = f"Message: {message}" # Prints: "Message: ..."
```

Example: Function With Default Arguments

```
# Function with default parameters
def greet(name="Guest"):
    return f"Hello, {name}!"

result1 = greet() # "Hello, Guest!"
result2 = greet("Alice") # "Hello, Alice!"
```

1.8 Managing Variables and Side Effects

- Python variables have different scopes:
  - **Local Scope**: Variables declared inside a function are local to that function.
  - **Global Scope**: Variables declared outside functions are accessible globally.
  - **Explicit Global Declaration**: Use the `global` keyword to modify global variables inside a function.
- Improper scoping can lead to unexpected side effects, such as unintended modifications to global variables.

Example: Local Scope

```python
# Local variable inside a function
def local_scope():
    local_var = 10
    # local_var is accessible only within this function
    result = local_var # 10
```

Example: Global Scope

```python
# Global variable
global_var = 20

def access_global():
    result = global_var # 20 (accessing the global variable)
```

Example: Modifying a Global Variable Using `global`

```python
# Global variable
counter = 0

def increment_counter():
    global counter
    counter += 1 # Modifies the global variable
    # counter is now incremented globally
```

Example: Possible Side Effects Without Proper Scoping

```
# Unintended modification of a global variable
value = 50

def modify_value():
    # Unintended: creates a local variable instead of
        modifying the global one
    value = 100 # This does NOT modify the global 'value'

modify_value()
result = value # 50 (global 'value' remains unchanged)
```

Example: Preventing Side Effects by Properly Using `global`

```
# Global variable
value = 50

def modify_global_value():
    global value
    value = 100 # Explicitly modifies the global 'value'

modify_global_value()
result = value # 100 (global 'value' is updated)
```

1.9 Object-Oriented Programming (OOP)

- Python supports Object-Oriented Programming (OOP) using classes and objects.
- **Classes** define objects with attributes and methods.
- Use the `__init__` method to initialize object attributes.
- **Inheritance** allows child classes to extend or override the functionality of parent classes.

Example: Defining a Class With Attributes and Methods

```python
# Class definition
class Animal:
    def __init__(self, name): # Initialize attributes
        self.name = name

    def speak(self): # Method
        return f"{self.name} makes a sound."

# Creating an object
dog = Animal("Dog")
result = dog.speak() # "Dog makes a sound."
```

Example: Inheritance

```python
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

# Child class inheriting from Animal
class Dog(Animal):
    def speak(self): # Overriding the parent method
        return f"{self.name} barks."

# Child class inheriting from Animal
class Cat(Animal):
    def speak(self): # Overriding the parent method
        return f"{self.name} meows."

# Creating objects
dog = Dog("Buddy")
cat = Cat("Whiskers")

result_dog = dog.speak() # "Buddy barks."
result_cat = cat.speak() # "Whiskers meows."
```

Example: Accessing Parent Methods in Child Classes

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound."

class Bird(Animal):
    def speak(self):
        parent_speak = super().speak() # Call parent method
        return f"{parent_speak} and sings."

# Creating an object
bird = Bird("Robin")
result = bird.speak() # "Robin makes a sound and sings."
```

1.10 Data Structures

1.10.1 Lists

- Lists are ordered, mutable, and allow duplicate values.
- Can store elements of different data types.
- Common operations:
  - `append(x)`: Add an item to the end.
  - `remove(x)`: Remove the first occurrence of a value.
  - `pop(i)`: Remove and return the item at index `i`.
  - `sort()`: Sort the list in ascending order.
  - `reverse()`: Reverse the list.

Example:

```
# Creating a list and performing operations
my_list = [3, 1, 4, 2]
my_list.append(5) # [3, 1, 4, 2, 5]
my_list.remove(1) # [3, 4, 2, 5]
my_list.sort()    # [2, 3, 4, 5]
```

1.10.2 Dictionaries

- Dictionaries store key-value pairs and are mutable.
- Keys must be unique and immutable, while values can be any data type.
- Common operations:
  - `get(key)`: Retrieve value for a key.
  - `keys()`: Get all keys.
  - `values()`: Get all values.
  - `del dict[key]`: Remove a key-value pair.
  - `in`: Check if a key exists.

Example:

```
# Creating a dictionary and performing operations
my_dict = {"a": 1, "b": 2, "c": 3}
my_dict["d"] = 4  # Add a new key-value pair
del my_dict["b"]  # {"a": 1, "c": 3, "d": 4}
value = my_dict.get("a") # 1
keys = list(my_dict.keys()) # ["a", "c", "d"]
```

1.10.3 Tuples

- Tuples are ordered, immutable, and allow duplicate values.
- Commonly used for fixed collections of data.
- Common operations:
  - Access elements using indices.
  - Use `count(x)` to count occurrences of a value.
  - Use `index(x)` to find the index of a value.

Example:

```python
# Creating a tuple and performing operations
my_tuple = (1, 2, 2, 3)
occurrences = my_tuple.count(2) # 2
index = my_tuple.index(3)     # 3
```

1.10.4 Arrays

- Arrays are similar to lists but are optimized for numerical computations.
- Use the `array` module or `numpy` for arrays.
- Common operations:
  - Indexing and slicing.
  - `append(x)`: Add a value at the end.
  - Arithmetic operations: Element-wise addition, subtraction, etc.

Example:

```python
import numpy as np
arr = np.array([1, 2, 3])
arr = arr + 2 # [3, 4, 5] (Element-wise addition)
```

1.10.5 Strings

- Strings are immutable sequences of characters.
- Common operations:
  - `len()`: Get the length of a string.
  - `strip()`: Remove whitespace.
  - `split()`: Split the string into a list.
  - `find(x)`: Find the first occurrence of a substring.
  - `replace(a, b)`: Replace occurrences of `a` with `b`.

Example:

```
text = " Hello, Python! "
text = text.strip() # "Hello, Python!"
words = text.split() # ["Hello,", "Python!"]
```

1.10.6 Deque (Double-Ended Queue)

- Deques are optimized for fast insertion and deletion from both ends.
- Use the `collections` module.
- Common operations:
  - `append(x)`: Add an item to the right end.
  - `appendleft(x)`: Add an item to the left end.
  - `pop()`: Remove and return an item from the right end.
  - `popleft()`: Remove and return an item from the left end.

Example:

```
from collections import deque
dq = deque([1, 2, 3])
dq.append(4)      # deque([1, 2, 3, 4])
dq.appendleft(0)  # deque([0, 1, 2, 3, 4])
dq.pop()          # deque([0, 1, 2, 3])
dq.popleft()      # deque([1, 2, 3])
```

1.10.7 Heap

- Heaps are binary trees used to implement priority queues.
- Use the `heapq` module for heaps in Python.
- Common operations:
  - `heapify(x)`: Convert a list into a heap.
  - `heappush(heap, x)`: Push an element into the heap.
  - `heappop(heap)`: Pop the smallest element from the heap.

Example:

```python
import heapq
heap = [5, 2, 3, 1]
heapq.heapify(heap)   # [1, 2, 3, 5]
heapq.heappush(heap, 0) # [0, 1, 3, 5, 2]
smallest = heapq.heappop(heap) # 0
```

1.11 File Reading and Writing

1.11.1 Text File Operations

- Different access modes for text files:
  - ’r’: Read-only mode (file must exist).
  - ’w’: Write mode (creates file or truncates existing file).
  - ’a’: Append mode (adds content to the end of the file).

Example: Creating and Writing to a Text File:

```python
# Write mode (’w’): Overwrites existing file or creates a new one
with open("example.txt", "w") as file:
    file.write("Hello, World!\n") # Creates ’example.txt’ with this text

# Append mode (’a’): Adds content to an existing file
with open("example.txt", "a") as file:
    file.write("Appending new content.\n")

# Read mode (’r’): Reads content of an existing file
with open("example.txt", "r") as file:
    content = file.read()
    # content: "Hello, World!\nAppending new content.\n"
```

1.11.2 CSV File Operations

- `csv` module simplifies reading and writing CSV files.
- Common access modes for CSV files:
  - ’r’: Read mode for reading rows.
  - ’w’: Write mode for creating and writing rows.
  - ’a’: Append mode for adding rows.

Example: Writing to and Reading from a CSV File:

```
import csv

# Writing to a CSV file
with open("example.csv", "w", newline="") as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(["Name", "Age", "City"]) # Adding header row
    writer.writerow(["Alice", 25, "New York"])
    writer.writerow(["Bob", 30, "Los Angeles"])

# Reading from a CSV file
with open("example.csv", "r") as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        # Each row is read as a list
        # Example: ["Name", "Age", "City"]
        pass
```

1.12 Exception Handling

1.12.1 Handling Exceptions in Python

- Exceptions occur when an error disrupts program execution.
- Use `try-except` blocks to handle exceptions gracefully.
- Common exceptions include:
  - `FileNotFoundError`: File does not exist.
  - `ZeroDivisionError`: Division by zero.
  - `ValueError`: Invalid value for a function.
  - `KeyError`: Accessing a non-existent dictionary key.

Example: Handling FileNotFoundError:

```python
try:
    with open("nonexistent.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    # File does not exist
    content = "Default content"
```

Example: Handling Division by Zero:

```python
try:
    result = 10 / 0
except ZeroDivisionError:
    result = None # Handle division by zero gracefully
```

Example: Handling Multiple Exceptions:

```python
try:
    value = int("not_a_number") # Raises ValueError
except ValueError:
    value = 0 # Handle invalid conversion
except KeyError:
    value = -1 # Handle dictionary key errors
```

Example: Using `else` and `finally`:

```python
try:
    num = int("42") # Conversion succeeds
except ValueError:
    num = 0
else:
    # Executed if no exceptions occurred
    num += 1 # num = 43
finally:
    # Executed regardless of exception occurrence
    print("Execution complete")
```