PYTHON PROGRAMMING

BCA IV SEM(NEP)

NOTES

Prepared By

Mr. Prabhu Kichadi, BE, MTECH

KLES SSMS BCA College Athani

UNIT – II

Exception Handling: Types of Errors; Exceptions; Exception Handling using try, except and finally.

Python Functions: Types of Functions; Function Definition- Syntax, Function Calling, Passing Parameters/arguments, the return statement; Default Parameters; Command line Arguments; Key Word Arguments; Recursive Functions; Scope and Lifetime of Variables in Functions.

Strings: Creating and Storing Strings; Accessing Sting Characters; the str() function; Operations on Strings- Concatenation, Comparison, Slicing and Joining, Traversing; Format Specifiers; Escape Sequences; Raw and Unicode Strings; Python String Methods.

Exception Handling: Exception handling is a mechanism in Python that allows you to catch and handle errors that occur during program execution. When an error occurs, an exception is raised, and if the exception is not handled, the program will terminate with an error message and program terminates abnormally.

Error: an error is an exceptional event or a mistake in a program that occurs during program execution and prevents the program from running as intended. Errors can occur for a variety of reasons, such as syntax errors, runtime errors, and logical errors.

Types of Errors:

1. Syntax Errors: Syntax errors occur when you violate the rules of the Python language. For example, forgetting to close a parenthesis or using an incorrect keyword can result in a syntax error.

2. Runtime Errors(Exceptions): Runtime errors occur during program execution when a statement attempts an operation that is impossible to carry out. Examples include division by zero or trying to access a variable that has not been defined.

3. Logical Errors: Logical errors occur when the program runs without raising an error, but the output is not what you intended.

Exceptions: an exception is an object that represents an error condition or unexpected situation that occurs during program execution. Exceptions are raised by the Python interpreter when it encounters an error that it cannot handle and terminates program abnormally.

When an exception is raised, the Python interpreter creates an exception object that contains information about the error, such as the type of error and a description of the error. The exception object is then passed up the call stack until it is either caught by an exception handler or the program terminates with an error message.

Exception Handling: Exception handling in Python is a mechanism for handling errors that may occur during program execution. When an error occurs in a program, an exception is raised and if the exception is not handled, the program will terminate with an error message.

The basic idea behind exception handling is to try to execute a piece of code that may raise an exception, and then catch and handle the exception if it is raised.

Exception Handling using try, except and finally:

- The **try and except** statements are used to implement exception handling in Python.
- **The try** block contains the code that may raise an exception, and
- the **except block** contains the code that should be executed if an exception is raised.
- **else block:** This block contains the code that you want to execute if no exceptions occur within the try block.
- **finally block:** This block contains the code that you want to execute regardless of whether an exception occurred or not

You can use try, except, else, and finally blocks in Python to handle exceptions and control the flow of your program. Here's how you can use each of these blocks:

- try block: This block contains the code that you want to execute. If an exception occurs within this block, the corresponding except block will be executed.

- except block: This block contains the code that you want to execute if an exception occurs within the try block. You can specify the type of exception that you want to catch using the except keyword followed by the exception type.

- else block: This block contains the code that you want to execute if no exceptions occur within the try block. It is optional and is executed after the try block and before the finally block.

- finally block: This block contains the code that you want to execute regardless of whether an exception occurred or not. It is executed after the try block and any except or else blocks.

Examples:

1. ZeroDivisionError: ZeroDivisionError is a type of exception that is raised when a program tries to divide a number by zero. In Python, dividing a number by zero is an undefined operation, and therefore results in a ZeroDivisionError.

Without handler:

```
a = 10
b = 0
c = a / b
print(c)
```

Console Error:

```
c = a / b
ZeroDivisionError: division by zero
```

Exception Handling using try, except, else and finally block:

```
a = 10
b = 0
c = 0
try:
   c = a / b
except ZeroDivisionError:
   print("Cant Divide by Zero...")
else:
   print("Result = ",c)
finally:
   print("Clean the resources from try")
```

Output:

```
Cant Divide by Zero...
Clean the resources from try
```

Built-in Exceptions in python:

In Python, there are many built-in types of exceptions that can be raised during the execution of a program. Here are some common types of exceptions in Python:

- SyntaxError: Raised when there is a syntax error in the code.
- TypeError: Raised when an operation or function is applied to an object of inappropriate type.
- NameError: Raised when a variable or name is not found in the current scope.
- AttributeError: Raised when an attribute reference or assignment fails.
- ValueError: Raised when a function or operation receives an argument of the correct type but with an inappropriate value.
- KeyError: Raised when a dictionary key is not found in the set of existing keys.
- IndexError: Raised when an index is not found in a sequence or list.
- ZeroDivisionError: Raised when the second argument of a division or modulo operation is zero.
- IOError: Raised when an input/output operation fails, such as when a file cannot be opened.
- ImportError: Raised when a module, package, or object cannot be imported.

Python Functions: A function is a block of reusable code that performs a specific task. It is defined using the def keyword, followed by the name of the function and a set of parentheses that may contain parameters or arguments.

The function block is indented and can contain any number of statements, including return statements.

Types of Functions: In Python, there are several types of functions, each with a specific purpose. Here are some of the most common types:

Built-in Functions: These are functions that are built into Python and can be used directly without any import statement. Examples include print(), len(), type(), etc.

User-defined Functions: These are functions that are defined by the user to perform a specific task. We define these functions using the def keyword.

Anonymous Functions (Lambda Functions): These are small, single-expression functions that do not have a name. They are defined using the lambda keyword and can be used wherever a function object is required.

Recursive Functions: These are functions that call themselves to solve a problem. They are often used to solve problems that can be broken down into smaller sub-problems.

Function Definition: In Python, a function is defined using the def keyword, followed by the name of the function and a set of parentheses that may contain parameters or arguments. The function block is indented and can contain any number of statements, including return statements.

Here is the syntax of a basic function in Python:

```
def function_name(parameter1, parameter2, ...):
    # function block
    statement1
    statement2
    ...
    return value
```

- In this syntax, function_name is the name of the function, and parameter1, parameter2, etc. are the parameters or arguments that the function takes. These parameters are optional and can be left empty if the function does not require any input.
- Inside the function block, we can include any number of statements that perform a specific task. These statements can be simple or complex, and can include loops, conditions, and other functions.

- The return statement is optional and is used to return a value from the function. If the function does not include a return statement, it will return None by default.
- Function defn must be written before program start.

Example:

```
def add(num1, num2):        # Function Definition
   sum = num1 + num2
   return sum

a = 10
b = 20
res = add(a,b)              # Function Call & Passing arguments
print("Result from add = ",res)
```

Passing Parameters/arguments: you can pass parameters or arguments to a function by including them inside the parentheses when you define the function. These parameters can then be used inside the function block to perform specific operations or calculations.

In this example, num1 and num2 are the two parameters that are passed to the function. Inside the function block, we add these two numbers together and store the result in a variable called sum. We then return this value using the return keyword.

```
def add(num1, num2):        # Function Definition
      sum = num1 + num2
      return sum

a = 10
b = 20
res = add(a,b)              # Function Call & Passing arguments
print("Result from add = ",res)
```

To call this function and pass arguments, you simply include the values inside the parentheses:

```
a = 33
b = 22
result = add(a,b)
```

In this example, we are calling the add_numbers function and passing the values a and b as arguments. The function will add these two numbers together and return the result, which will be stored in the variable result.

The return statement: return statement is used to exit a function and return a value or expression to the caller. The return statement can be used in any function, regardless of its parameters or body.

The return statement can also be used to return multiple values from a function, using tuples or other data structures. Here's an example:

```
def calculate_stats(numbers):
    total = sum(numbers)
    average = total / len(numbers)
    return total, average

data = [10, 20, 30, 40, 50]
total, average = calculate_stats(data)
print(total)    # Output: 150
print(average)  # Output: 30.0
```

Default Parameters: In Python, you can define default parameter values for function arguments. This allows you to provide a default value for an argument that will be used if no value is passed in when the function is called.

Example:

```
def greet(name='John'):
        print(f"Hello, {name}!")

greet()       # Output: Hello, John!
greet("Van")  # Output: Hello, Van
```

Command line Arguments: Command line arguments are arguments passed to a Python script(Program) when it is executed from the command line.

In Python, you can access command line arguments using **the sys.argv** list. The sys module provides access to some variables used or maintained by the interpreter, and one of these variables is argv. The argv list contains the command line arguments passed to the script, with the first argument being the name of the script itself.

Example:

```
import sys

# Print the command line arguments
print(sys.argv)

# Print the script name and the first argument
```

```
print("Script name:", sys.argv[0])
print("First argument:", sys.argv[1])
```

How to execute and pass command line arguments:

```
C:\Users\Prabhu Kichadi\OneDrive\Desktop\Python>py sample.py hello
['sample.py', 'hello']
Script name: sample.py
First argument: hello
```

Key Word Arguments: Keyword arguments are a way to pass arguments to a Python function by specifying the name of the argument with its corresponding value. In other words, instead of passing the values in a strict positional order, you can pass them in any order as long as you specify the argument name.

you can pass keyword arguments to a function using the syntax key=value. Keyword arguments are useful when you want to specify arguments out of order or when you want to provide default values for some arguments.

Example:

```
def greet(name, message="Hello"):
    print(message, name)

# Call the greet() function with keyword arguments
greet(name="Alice")
greet(message="Hi", name="Bob")
```

Recursive Functions: Recursive functions in Python are functions that call themselves in order to solve a problem.

A recursive function typically has a base case and a recursive case. The base case is a condition that terminates the recursion, while the recursive case calls the function again with a simplified version of the original problem.

Example:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

In this example, the factorial function calculates the factorial of a number n. If n is zero, the function returns 1 (the base case). Otherwise, the function returns n times the factorial of n-1 (the recursive case). For example, factorial(5) returns 5 * 4 * 3 * 2 * 1 = 120.

Scope and Lifetime of Variables in Functions:

Scoping: scoping refers to the rules that determine how variables are accessed and modified within a program. Indicates life and scope of a variables.

The scope of a variable refers to the region of the code where the variable is accessible.

The lifetime of a variable refers to the period of time during which the variable exists in memory.

Local Scope: Variables declared inside a function have local scope. This means that they are only accessible within the function and do not exist outside of it.

Local scope refers to variables that are defined inside a function or class. These variables can only be accessed from within the function or class in which they are defined. If a variable with the same name is defined both locally and globally, the local variable takes precedence over the global variable.

For example:

```
def my_function():
    x = 10
    print(x)

my_function()  # Output: 10
print(x)  # NameError: name 'x' is not defined
```

In this example, the variable x is declared inside the my_function function and is only accessible within it. When we try to print the value of x outside of the function, we get a NameError because the variable does not exist in that scope.

Global Scope: If a variable is declared outside of the function, it has a global scope and can be accessed from within the function: Global scope refers to variables that are defined outside of any function or class. These variables can be accessed from anywhere in the program, including inside functions or classes.

Example:

```
x = 5

def my_function():
  print(x)

my_function()  # Output: 5
```

Example: Local and Global Scooping:

```
x = 5                # Global Variable

def my_function():
   z = 33          # Local Variable
   print("Global Variable Inside Function = ",x)
   print("Local Variable = ",z)

my_function()  # Output: 5
print("Global Variable Outside Function = ",x)
```

Strings: A string is a sequence of characters. It is a data type that is used to represent textual data, such as words or sentences.

Strings are created using either single quotes ('...') or double quotes ("..."), and they can contain any combination of letters, digits, and special characters.

Strings are also immutable, which means that once a string is created, it cannot be modified.

Creating and Storing Strings: you can create and store strings in various ways. Here are a few examples:

1. Using string literals: You can create a string by enclosing a sequence of characters in either single quotes ('...') or double quotes ("..."). For example:

```
string1 = 'Hello, world!'
string2 = "This is a string."
```

2. Using the str() function: You can convert a value to a string using the str() function. For example:

```
num = 42
string3 = str(num)
```

Accessing Sting Characters: you can access individual characters in a string using indexing. The index of the first character in a string is 0, and the index of the last character is len(string) - 1.

There are several techniques for accessing string characters in Python. Here are some of the most common ones:

1. Indexing: You can access an individual character in a string using indexing. The index of the first character is 0, and the index of the last character is len(string)-1.

Example:

```
string = "hello"; print(string[0]) # Output: 'h'
```

2. Slicing: You can extract a substring from a string using slicing. The syntax for slicing is

Slicing Syntax: string[start: end: step],

where start is the index of the first character to include, end is the index of the first character to exclude, and step is the increment between characters.

Example:

```
string = "hello";
print(string[1:4]) # Output: 'ell'
```

3. Negative indexing: You can access characters from the end of a string using negative indexing. The index of the last character is -1, and the index of the first character is -len(string).

Example:

```
string = "hello";
print(string[-1]) # Output: 'o'
```

4. Iteration: You can iterate over a string using a for loop to access each character in turn.

Example:

```
string = "hello";
for char in string:
        print(char)
```

The str() function: the str() function is used to convert an object into a string. It can be used to convert objects of different data types, such as integers, floating-point numbers, lists, and tuples, into a string.

The syntax of the str() function is as follows:

```
str(object)
```

Example:

```
# Converting an integer into a string
x = 10
y = str(x)
print(y)   # Output: "10"
print(type(y))
```

```
# Converting a floating-point number into a string
z = 3.14
w = str(z)
print(w)   # Output: "3.14"

# Converting a list into a string
my_list = [1, 2, 3]
list_str = str(my_list)
print(list_str)   # Output: "[1, 2, 3]"

# Converting a tuple into a string
my_tuple = (4, 5, 6)
tuple_str = str(my_tuple)
print(tuple_str)  # Output: "(4, 5, 6)"
```


Operations on Strings- Concatenation, Comparison, Slicing and Joining, Traversing; Format Specifiers; Escape Sequences; Raw and Unicode Strings;

1. Concatenation: In Python, the + operator is used for string concatenation. It allows you to join two or more strings into a single string. Here is an example:

Ex:

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result)  # Output: "Hello World"
```

2. Comparison: Python provides several operators for string comparison, including ==, !=, <, >, <=, and >=. These operators compare two strings lexicographically, character by character. For example:

```
str1 = "apple"
str2 = "banana"
print(str1 == str2)  # Output: False
print(str1 < str2)   # Output: True
```

3. Slicing and Joining: You can extract a substring from a string by using the slicing operator : You can also join two or more strings into a single string using the join() method. Here are some examples:

Example:

```
# Slicing
str1 = "Hello World"
substring = str1[0:5]  # Extract the first five characters
print(substring)      # Output: "Hello"

# Joining
list_of_strings = ["apple", "banana", "cherry"]
separator = ", "
result = separator.join(list_of_strings)
print(result)        # Output: "apple, banana, cherry"
```

4. Traversing: You can traverse a string using a loop or a list comprehension. Here is an example:

```
str1 = "Hello World"
for character in str1:
   print(character)
```

5. Format Specifiers: You can use format specifiers to format a string in a specific way. The % operator is used to format a string. Here is an example:

```
name = "John"
age = 30
print("My name is %s and I am %d years old." % (name, age))
```

6. Escape Sequences: Escape sequences are used to represent special characters in a string. They are preceded by a backslash (\). Here are some common escape sequences:

Escape Sequence	Description
`\n`	Newline
`\t`	Tab
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash

7. Raw and Unicode Strings: In Python, you can create raw strings by prefixing the string literal with an r. This means that escape sequences will be ignored. You can also create Unicode strings by prefixing the string literal with a u. Here are some examples:

```
# Raw string
str1 = r"C:\Users\John\Documents"
print(str1)  # Output: "C:\Users\John\Documents"

# Unicode string
str2 = u"\u03a3"
print(str2)  # Output: "Σ"
```

Python String Methods. Python provides various built-in methods to manipulate strings. Some of the commonly used string methods in Python are:

1. **strip():** This method is used to remove any leading or trailing whitespace characters from a string.

   Ex:

   ```
   text = "     hello world     "
   print(text.strip()) # Output: "hello world"
   ```

2. **lower():** This method is used to convert all characters in a string to lowercase.

   Ex:

   ```
   text = "Hello World"
   print(text.lower()) # Output: "hello world"
   ```

3. **upper():** This method is used to convert all characters in a string to uppercase.

   Ex:

   ```
   text = "Hello World"
   print(text.upper()) # Output: "HELLO WORLD"
   ```

4. **replace():** This method is used to replace a specified substring with another substring in a string.

   Ex:

   ```
   text = "Hello World"
   print(text.replace("World", "Python")) # Output: "Hello Python"
   ```

5. **split():** This method is used to split a string into a list of substrings based on a specified delimiter.

   Ex:

```
text = "Hello,World"
print(text.split(",")) # Output: ['Hello', 'World']
```

6. **join():** This method is used to join a list of strings into a single string using a specified delimiter.

   Ex:

```
my_list = ['Hello', 'World']
print('-'.join(my_list)) # Output: "Hello-World"
```

7. **find():** This method is used to find the index of the first occurrence of a specified substring in a string.

   Ex:

```
text = "Hello World"
print(text.find("o")) # Output: 4
```

8. **count():** This method is used to count the number of occurrences of a specified substring in a string.

   Ex:

```
text = "Hello World"
print(text.count("l")) # Output: 3
```

9. **startswith():** This method is used to check if a string starts with a specified substring.
   Ex:

```
text = "Hello World"
print(text.startswith("H")) # Output: True
```

10. **endswith():** This method is used to check if a string ends with a specified substring.

    Ex:

```
text = "Hello World"
print(text.endswith("d")) # Output: True
```

11. isalpha(): This method is used to check if all characters in a string are alphabets.

Ex:

```
text = "Hello World"
print(text.isalpha()) # Output: False
```

12. isdigit(): This method is used to check if all characters in a string are digits.

Ex:

```
text = "123"
print(text.isdigit()) # Output: True
```

13. capitalize(): capitalize() is a string method that returns a copy of the original string with the first character capitalized and the rest of the characters in lowercase.

Ex:

```
text = "hello world"
print(text.capitalize()) # Output: FHello world
```

14. isalpnum(): This method is used to check if all characters in a string are alphabets or numbers.

Ex:

```
text = "Hello"
print(text.isalnum()) # Output: True
```

Note: Refer all python Lab journal programs on this unit.