PYTHON PROGRAMMING (20MCA21C)

UNIT - IV

Tuples, Dictionaries and Exceptions

FACULTY:

Dr. R. A. Roseline, M.Sc., M.Phil., Ph.D.,

Associate Professor and Head,
Post Graduate and Research Department of Computer Applications,
Government Arts College (Autonomous), Coimbatore - 641 018.

Tuples

TUPLES

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional.

A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Different types of tuples
 # Empty tuple
 my_tup = ()
print(my_tup)
# Tuple having integers
my_tup = (10, 2, 33)
print(my_tup)
# tuple with mixed datatypes
my_tup = (1, “PYTHON", 3.89)
print(my_tup)
# nested tuple
my_tup = (“MCA", [1, 4, 9], (1, 2, 3))
print(my_tup)
```

Tuple packing.

This is known as tuple packing.

```
my_tup = 2, 14.6, “MCA“
print(my_tup)
 # tuple unpacking is also possible
a, b, c = my_tup
print(a)

print(b)

print(c)
```

Creating a tuple with one element

We will need a trailing comma to indicate that it is, in fact, a tuple.

```
my_tup = (“MCA")
print(type(my_tup))
# <class 'str'>

 # Creating a tuple having one element
my_tuple = ("hello",)
print(type(my_tuple))
# <class 'tuple'>

# Parentheses is optional
my_tuple = "hello",
print(type(my_tuple))
# <class 'tuple'>
```

Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

We can use the index operator [] to access
 an item in a tuple, where the index starts from 0.
So, a tuple having 10 elements will have indices from 0 to 9.
Trying to access an index outside of the tuple
index range(10,11,... in this example) will raise an IndexError.
The index must be an integer, so we cannot use float or other types.
This will result in TypeError.
Likewise, nested tuples are accessed using nested
indexing, as shown in the example below.

```
# Accessing tuple elements using indexing
my_tup = (‘M','C',A')
print(my_tuple[0])
print(my_tuple[5])
# 't' # IndexError: list index out of range
```

DICTIONARIES

Creating a dictionary

{} separated by commas.
An item has a **key** and a corresponding **value** that is expressed as a pair (key: value).

While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
# empty dictionary
 my_dict = {}
# dictionary with integer keys
my_dict = {1: ‘App', 2: 'bill'}
# dictionary with mixed keys
 my_dict = {'name': ‘Ash’, 1: [2, 4, 3]}

 # using dict()
 my_dict = dict({1:Art', 2:'bill'})

 # from sequence having each item as a pair
 my_dict = dict([(1,‘Art'), (2,'bill')])
```

As you can see from above, we can also create a dictionary using the built-in **dict()** function.

While indexing is used with other data types to access values,
a dictionary uses keys.
Keys can be used either inside square brackets [] or with the get() method.
If we use the square brackets [],
KeyError is raised in case a key is not found in the dictionary.
On the other hand, the get() method returns None if the key is not found.

```
# get vs [] for retrieving elements
 my_dict = {'name': 'Ann', 'age': 16}
 print(my_dict['name'])
print(my_dict.get('age'))
# Trying to access keys which doesn't exist throws error
# Output None
print(my_dict.get('address'))
# KeyError
 print(my_dict['address'])
```

Changing and Adding Dictionary elements

We can add new items or change the value
of existing items using an assignment operator.
If the key is already present, then the existing value gets updated
. In case the key is not present, a new (**key: value**) pair is added to the dictionary.

```
# Changing and adding Dictionary Elements
 my_dict = {'name': ‘Ann', 'age': 16}
# update value
my_dict['age'] = 20
#Output: {'age': 20, 'name': ‘Ann'}
print(my_dict)
# add item my_dict['address'] = ‘Coimbatore'

print(my_dict)
```

Removing elements from Dictionary

pop() method.

This method removes an item with the provided key and returns the value.
The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary.
All the items can be removed at once, using the clear() method.
We can also use the del keyword to remove individual items or the entire dictionary itself.

```
# Removing elements from a dictionary
# create a dictionary
squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
 # remove a particular item, returns its value
 # Output: 16
print(squares.pop(4))
 # Output: {1: 1, 2: 4, 3: 9, 5: 25}
print(squares)
 # remove an arbitrary item, return (key,value)
 # Output: (5, 25)
 print(squares.popitem())
# Output: {1: 1, 2: 4, 3: 9}
print(squares)
# remove all items
squares.clear()
# Output: {}
print(squares)
# delete the dictionary itself
del squares
 # Throws Error
```

Python Dictionary Methods

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Returns the value of the key. If the key does not exist, returns d (defaults to None).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the key and returns its value or d if key is not found. If d is not provided and the key is not found, it raises KeyError.
popitem()	Removes and returns an arbitrary item (**key, value**). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the key is in the dictionary. If not, inserts the key with a value of d and returns d (defaults to None).
update([other])	Updates the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Returns a new object of the dictionary's values

Dictionary Membership Test

key is in a dictionary or not using the keyword in.
The **membership test is only for the keys and not for the values**.

```
# Membership Test for Dictionary Keys

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
# Output: True
 print(1 in squares)
# Output: True
 print(2 not in squares)

# membership tests for key only not value
 # Output: False
 print(49 in squares)
```

Dictionary Built-in Functions

Built-in functions like all(), any(), len(), cmp(), sorted(),
etc. are commonly used with dictionaries to perform different tasks.

	Description
all()	Return True if all keys of the dictionary are True (or if the dictionary is empty).
any()	Return True if any key of the dictionary is true. If the dictionary is empty, return False.
len()	Return the length (the number of items) in the dictionary.
cmp()	Compares items of two dictionaries. (Not available in Python 3)
sorted()	Return a new sorted list of keys in the dictionary.

EXCEPTIONS

Exception Handling –

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them –

Here is a list standard Exceptions available in Python:

Sr.No.	Exception Name & Description
1	Exception Base class for all exceptions
2	StopIteration Raised when the next() method of an iterator does not point to any object.
3	SystemExit Raised by the sys.exit() function.
4	StandardError Base class for all built-in exceptions except StopIteration and SystemExit.
5	ArithmeticError Base class for all errors that occur for numeric calculation.
6	OverflowError Raised when a calculation exceeds maximum limit for a numeric type.
7	FloatingPointError Raised when a floating point calculation fails.
8	ZeroDivisionError Raised when division or modulo by zero takes place for all numeric types.
9	AssertionError Raised in case of failure of the Assert statement.
10	AttributeError Raised in case of failure of attribute reference or assignment.
11	EOFError Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	ImportError Raised when an import statement fails.

13	KeyboardInterrupt Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	LookupError Base class for all lookup errors.
15	IndexError Raised when an index is not found in a sequence.
16	KeyError Raised when the specified key is not found in the dictionary.
17	NameError Raised when an identifier is not found in the local or global namespace.
18	UnboundLocalError Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	EnvironmentError Base class for all exceptions that occur outside the Python environment.
20	IOError Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	IOError Raised for operating system-related errors.
22	SyntaxError Raised when there is an error in Python syntax.
23	IndentationError Raised when indentation is not specified properly.
24	SystemError Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	SystemExit Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	TypeError Raised when an operation or function is attempted that is invalid for the specified data type.
27	ValueError Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	RuntimeError Raised when a generated error does not fall into any category.
29	NotImplementedError Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

What is Exception?

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
- Handling an exception
- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.
- Syntax

In Python, exceptions can be handled using a try statement.
The critical operation which can raise an exception is placed inside the try clause.
The code that handles the exceptions is written in the except clause.
We can thus choose what operations to perform once we have caught the exception.
Here is a simple example.

```
# import module sys to get the type of exception
import sys randomList = ['a', 0, 4]
for entry in randomList:
    try: print("The entry is", entry)
         r = 1/int(entry)
         break
    except: print("ERROR!", sys.exc_info()[0], "occurred.")
         print("Next entry.")
         print()
print("The reciprocal of", entry, "is", r)
```

```
The entry is a ERROR! <class 'ValueError'> occurred.
Next entry.
The entry is 0 ERROR! <class 'ZeroDivisionError'> occured.
Next entry.
 The entry is 2
The reciprocal of 4 is 0.25
```

In this program, we loop through the values of the randomList list.
As previously mentioned, the portion that can cause an exception is placed inside the try block.
If no exception occurs, the except block is skipped and normal flow continues(for last value).
But if any exception occurs, it is caught by the except block (first and second values).
Here, we print the name of the exception using the exc_info() function inside sys module.
We can see that a causes ValueError and 0 causes ZeroDivisionError.

rgument to an Exception in Python?

There might arise a situation where there is a need for additional information from an exception raised by Python.
Python has two types of exceptions namely, Built-In Exceptions and User-Defined Exceptions.

Why use Argument in Exceptions?

Using arguments for Exceptions in Python is useful for the following reasons:
•It can be used to gain additional information about the error encountered.ADVERTISING
•As contents of an Argument can vary depending upon different types of Exceptions in Python,
•Variables can be supplied to the Exceptions to capture the essence of the encountered errors.
• Same error can occur of different causes, Arguments helps us identify the
• specific cause for an error using the **except** clause.
•It can also be used to trap multiple exceptions, by using a variable to follow the tuple of Exceptions.

Arguments in Built-in Exceptions:

The below codes demonstrates use of Argument with Built-in Exceptions:

Example 1:

```
try:
    b = float(10 + 500 / 0)
except Exception as Argument:
    print( 'This is the Argument\n', Argument)
```

Output:

```
This is the Argument division by zero
```

User defined exceptions

- **Creating User-defined Exception**
- **Programmers may name their own exceptions by creating a new exception class. Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in "Error" similar to naming of the standard exceptions in python. For example:**

Example: In this example, we will illustrate how user-defined exceptions can be used in a program to raise and catch errors.
This program will ask the user to enter a number until they guess a stored number correctly.
To help them figure it out, a hint is provided whether their guess is greater than or less than the stored number.

```
# define Python user-defined exceptions
```

User-Defined Exception in Python

```
     Error(Exception):
"""Base class for other exceptions"""
          Pass
 class ValueTooSmallError(Error)
: """Raised when the input value is too small"""
          pass

class ValueTooLargeError(Error):
 """Raised when the input value is too large"""
    pass
# you need to guess this number
   number = 10
# user guesses a number until he/she gets it right
while True:
          try:
           i_num = int(input("Enter a number: "))
          if i_num < number:
                     raise ValueTooSmallError
          elif i_num > number:
                     raise ValueTooLargeError break
          except ValueTooSmallError:
                    print("This value is too small, try again!")
                    print()
          except ValueTooLargeError:
                     print("This value is too large, try again!")
                    print()
print("Congratulations! You guessed it correctly.")
```

Thank you

The Content in this Material are from the Textbooks and Reference books given in the Syllabus