**Instructions**: You must read the material and create an outline of the topics in your OWN words.  Do not copy the text from the tutorials into your notes. Make sure your outline contains notes for each subsection of the reading assignment. Thoroughly cover each topic to show you have a firm understanding of the programming concept or construct.

| **Ques** | **NOTES:** |
| --- | --- |
| Python’s Request Module | Response.json() – Python requests  Python request is generally used to fetch the content from a particular resource URL and returns a request object. In the request object, we can access the content, headers, and so on.  How to use response.json() using Python requests?  We need to install the requests modules to access their response.json() method.  Parsing Python requests Response JSON Content  As seen in the code below, we first fetch the data from an API using requests.get() method and store in a variable “response.” On printing the response returned we get ‘<Response [200]>’ which is the HTTP code that indicates success. To print the JSPO data fetched we use the json() method which prints the JSON data in the Python dictionary format as seen in the output and with this we can parse the JSON responses in Python.  # import requests module  **import** requests    # Making a get request  response **=** requests.get('[https://api.github.com](https://api.github.com/)')    # print response  **print**(response)    # print json content  print(response.json())  Output: |
| Python – Exceptions Handling | Convert Request Response to Dictionary in Python  We will parse the JSON data and print that data the same way we access the keys and values of a dictionary. After making the get request to an API, we store the JSON data in a variable “API\_Data” using the response.json() method. Now, we iterate over the JSON data using a for loop and print the data by using the keys.  # import requests module  **import** requests    # Making a get request  response **=** requests.get('[https://api.github.com](https://api.github.com/)')    # Store JSON data in API\_Data  API\_Data **=** response.json()    # Print json data using loop  **for** key **in** API\_Data:{      print(key,":", API\_Data[key])  }  Output:  How to Pretty Print a JSON Object From Python Requests  We will pretty print the JSON object that we got from an API using request.get() method. For that after converting the JSON object to the dictionary and stored into “response\_dict” we will apply the json.dumps() method on the data. Now we apply indentation on the data by passing the argument “indent=4” and sorting the keys by setting “sort\_keys=True” and then printing the data.  # import required modules  **import** requests  **import** json    # Making a get request  response **=** requests.get('[https://api.github.com](https://api.github.com/)')    # Convert json into dictionary  response\_dict **=** response.json()    # Pretty Printing JSON string back  **print**(json.dumps(response\_dict, indent**=**4, sort\_keys**=**True))  Output:  Advanced Concepts:  Many libraries are used to make HTTP requests, such as httplib, urlib, httplib2, treq, etc., but requests are one of the best with really cool features. If any attribute of requests shows NULL, check the status code using “requests.status\_code,” and if the status\_code doesn’t lie in the range of 200-29, then you need to check the method used to make the call and the URL you are requesting resources from.  This is defined as an event that occurs during the execution of a program and disrupts the normal flow of the program instructions. This could be caused by attempting to divide by zero, accessing a non-existent file, or trying to convert a string value to an integer where it’s not possible.  Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them –   * Exception Handling * Assertions   List of Standard Exceptions –   |  |  | | --- | --- | | **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. | |
|  | Assertions in Python  This is a sanity check that you can turn on or off when you are done with your program testing.  We can also look at it as a raise-if statement (or, more accurately, a raise-if-not statement.) Here, an expression is tested, and an exception is raised if the result turns out false.  Assert statements are used to carry out assertions and were introduced in Python version 1.5.  Programmers use it to check for valid input and valid outputs at the start and end of a program, respectively.  The assert Statement:  When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.  Syntax for assert is –  Assert Expresssion[, Arguments]  If the assertion fails, Python uses the supplied ArgumentExpression as the argument for the AssertionError, and they can be caught and handled like any other exception using the try-except statement. Still, if not handled, they will terminate the program and produce a traceback.  Example:  def KelvinToFahrenheit(Temperature):  assert (Temperature >= 0),"Colder than absolute zero!"  return ((Temperature-273)\*1.8)+32  print (KelvinToFahrenheit(273))  print (int(KelvinToFahrenheit(505.78)))  print (KelvinToFahrenheit(-5))  Output:  32.0  451  Traceback (most recent call last):  File "test.py", line 9, in <module>  print (KelvinToFahrenheit(-5))  File "test.py", line 4, in KelvinToFahrenheit  assert (Temperature >= 0),"Colder than absolute zero!"  AssertionError: Colder than absolute zero!  What is An Exception?  This event occurs during the execution of a program that disrupts the normal flow of the program’s instructions. Generally, in a Python script, an exception is raised if a situation cannot be coped with. It is a Python object that represents an error.  Handling an Exception in Python  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.   * The **try**: block contains statements which are susceptible for exception * If exception occurs, the program jumps to the **except**: block. * If no exception in the **try**: block, the **except**: block is skipped.   Syntax:  try:  You do your operations here  ......................  except ExceptionI:  If there is ExceptionI, then execute this block.  except ExceptionII:  If there is ExceptionII, then execute this block.  ......................  else:  If there is no exception then execute this block.  Take note of the following:   * A single **try** statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions. * You can also provide a generic **except** clause, which handles any exception. * After the except clause(s), you can include an **else** clause. The code in the **else** block executes if the code in the try: block does not raise an exception. * The **else** block is a good place for code that does not need the try: block's protection.   Example:  try:  fh = open("testfile", "w")  fh.write("This is my test file for exception handling!!")  except IOError:  print ("Error: can\'t find file or read data")  else:  print ("Written content in the file successfully")  fh.close()  Output:  Written content in the file successfully  But if we change the mode parameter in open() function to “r” and for this, the testfile is not already present, the program encounters IOError in the except block and prints the following error message –  Error: can’t find file or read data  Example:  This example tries to open a file where you do not have write permission, so it raises an exception –  try:  fh = open("testfile", "r")  fh.write("This is my test file for exception handling!!")  except IOError:  print ("Error: can\'t find file or read data")  else:  print ("Written content in the file successfully")  This produces the following result –  Error: can’t find file or read data  The except Clause with No Exceptions  try:  You do your operations here;  ......................  except:  If there is any exception, then execute this block.  ......................  else:  If there is no exception then execute this block.  Here, the try-except statement catches all the exceptions that occur. This is not considered a good practice as it catches all exceptions but does not allow the programmer to identify the root cause of the problem that may occur.  The except Clause with Multiple Exceptions  try:  You do your operations here;  ......................  except(Exception1[, Exception2[,...ExceptionN]]]):  If there is any exception from the given exception list,  then execute this block.  ......................  else:  If there is no exception then execute this block.  The try-finally Clause  You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not.  try:  You do your operations here;  ......................  Due to any exception, this may be skipped.  finally:  This would always be executed.  ......................  Note that you cannot use else clause as well along with a finally clause.  Example:  try:  fh = open("testfile", "w")  fh.write("This is my test file for exception handling!!")  finally:  print ("Error: can\'t find file or read data")  This can also be written as thus:  try:  fh = open("testfile", "w")  try:  fh.write("This is my test file for exception handling!!")  finally:  print ("Going to close the file")  fh.close()  except IOError:  print ("Error: can\'t find file or read data")  When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the *try-except* statement.  Argument of an Exception  Exception can have arguments whose value can give additional information about the problem. The contents of the argument vary by exception.  try:  You do your operations here;  ......................  except ExceptionType, Argument:  You can print value of Argument here...  If you write the code to handle a single exception, you can have a variable that follows the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.  This variable receives the exception value, which mostly contains the exception's cause. The variable can receive a single value or multiple values as a tuple. This tuple usually contains the error string, the error number, and an error location.  # Define a function here.  def temp\_convert(var):  try:  return int(var)  except ValueError as Argument:  print ("The argument does not contain numbers\n", Argument)  # Call above function here.  temp\_convert("xyz")  Output:  The argument does not contain numbers  invalid literal for int() with base 10: 'xyz'  Raising Exceptions  The general syntax for doing this is as follows:  Raise [Exception [, args [, traceback]]]  Exception is the exception type (for example, NameError), and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None. The final argument, trackback, is also optional (and rarely used in practice) and, if present, is the traceback object used for the exception.  Example:  An exception can be strings, a class, or an object. Most of the exceptions the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows –  def functionName( level ):  if level < 1:  raise "Invalid level!", level  # The code below to this would not be executed  # if we raise the exception  We can use this code example to catch this newly defined exception:  try:  Business Logic here...  except "Invalid level!":  Exception handling here...  else:  Rest of the code here...  Note the raise name and the except name used.  User-Defined Exceptions  We can create our own exceptions by deriving classes from the standard built-in exceptions.  In the below example, we subclassed the RuntimeError class:  class Networkerror(RuntimeError):  def \_\_init\_\_(self, arg):  self.args = arg  we use it like this:  try:  raise Networkerror("Bad hostname")  except Networkerror,e:  print (e.args) |