Python – S9

Contents

1. Exception
2. Try-except-finally and raise statements

Exceptions

Exceptions

- In Python, exceptions are triggered automatically on errors, and they can be triggered and intercepted by your code.
- A well-written program should produce valuable results even when exceptional conditions occur.
- A program depends on numerous resources: memory, files, other packages, input-output devices, to name a few.
- Sometimes it is best to treat a problem with any of these resources as an exception, which interrupts the normal sequential flow of the program.
- An exception is an event that interrupts the ordinary sequential processing of a program.
- When an exception is raised, Python will handle it immediately.
- Python does this by examining except clauses associated with try statements to locate a suite of statements that can process the exception.
- If there is no except clause to handle the exception, the program stops running, and a message is displayed on the standard error file.

Exceptions

- An exception has two sides: the dynamic change to the sequence of execution and an object that contains information about the exceptional situation.
- The dynamic change is initiated by the raise statement, and can finish with the handlers that process the raised exception.
- If no handler matches the exception, the program's execution effectively stops at the point of the raise.

- In addition to the dynamic side of an exception, an object is created by the raise statement; this is used to carry any information associated with the exception.

Exceptions

- Consequences. The use of exceptions has two important consequences.

- First, we need to clarify where exceptions can be raised.
- Since various places in a program will raise exceptions, and these can be hidden deep within a function or class, their presence should be announced by specifying the possible exceptions in the docstring.

- Second, multiple parts of a program will have handlers to cope with various exceptions.
- These handlers should handle just the meaningful exceptions.
- Some exceptions (like RuntimeError or MemoryError) generally can't be handled within a program; when these exceptions are raised, the program is so badly broken that there is no real recovery.

Exceptions

- Exceptions are a powerful tool for dealing with rare, atypical conditions.
- Generally, exceptions should be considered as different from the expected or ordinary conditions that a program handles.
- For example, if a program accepts input from a person, exception processing is not appropriate for validating their inputs.
- There's nothing rare or uncommon about a person making mistakes while attempting to enter numbers or dates.
- On the other hand, an unexpected disconnection from a network service is a good candidate for an exception; this is a rare and atypical situation.
- Examples of good exceptions are those which are raised in response to problems with physical resources like files and networks.
- Python has a large number of built-in exceptions, and a programmer can create new exceptions.
- Generally, it is better to create new exceptions rather than attempt to stretch or bend the meaning of existing exceptions.

Exception Handling

- Python uses special objects called *exceptions* to manage errors that arise during a program's execution.
- Whenever an error occurs that makes Python unsure what to do next, it creates an exception object.
- If you write code that handles the exception, the program will continue running.
- If you don't handle the exception, the program will halt and show a *traceback*, which includes a report of the exception that was raised.
- Exceptions are handled with try-except blocks.
- A try-except block asks Python to do something, but it also tells Python what to do if an exception is raised.
- When you use try-except blocks, your programs will continue running even if things start to go wrong.
- Instead of tracebacks, which can be confusing for users to read, users will see friendly error messages that you write.

Exception Handling

- Let’s look at a simple error that causes Python to raise an exception. You probably know that it’s impossible to divide a number by zero, but let’s ask Python to do it anyway:

```
>>> print(5/0)
Traceback (most recent call last):
    File "division.py", line 1, in <module>
print(5/0)
ZeroDivisionError: division by zero
```

- The error reported at in the traceback, ZeroDivisionError, is an exception object.
- Python creates this kind of object in response to a situation where it can’t do what we ask it to.
- When this happens, Python stops the program and tells us the kind of exception that was raised.
- We can use this information to modify our program.
- We’ll tell Python what to do when this kind of exception occurs; that way, if it happens again, we’re prepared.

Exception Handling

- We can hence write a try-except block as :

```
try:
        print(5/0)
except ZeroDivisionError:
        print("You can't divide by zero!")
```

- So now we get the output as :

```
You can't divide by zero!
```

- If more code followed the try-except block, the program would continue running because we told Python how to handle the error.
- Let’s look at an example where catching an error can allow a program to continue running.

Exception Handling

```
# division.py
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    try:
        answer = int(first_number) / int(second_number)
    except ZeroDivisionError:
        print(“You can’t divide by 0 !!”)
    else:
        print(answer)
```

Exception Handling

- The try-except-else block works like this: Python attempts to run the code in the try statement. The only code that should go in a try statement is code that might cause an exception to be raised. Sometimes you'll have additional code that should run only if the try block was successful; this code goes in the else block. The except block tells Python what to do in case a certain exception arises when it tries to run the code in the try statement.

- By anticipating likely sources of errors, you can write robust programs that continue to run even when they encounter invalid data and missing resources.
- Your code will be resistant to innocent user mistakes and malicious attacks.

- One common issue when working with files is handling missing files.
- The file you're looking for might be in a different location, the filename may be misspelled, or the file may not exist at all.
- You can handle all of these situations in a straightforward way with a try-except block.

Exception Handling

```
filename = 'alice.txt'
try:
        with open(filename) as f_obj:
                contents = f_obj.read()
except FileNotFoundError:
        msg = "Sorry, the file " + filename + " does not exist."
        print(msg)
```

- The program has nothing more to do if the file doesn’t exist, so the error-handling code doesn’t add much to this program.
- Let’s build on this example and see how exception handling can help when you’re working with more than one file.

Exception Handling

Analyzing Text

- You can analyze text files containing entire books.
- Many classic works of literature are available as simple text files because they are in the public domain. The texts used in this section come from Project Gutenberg (*http://gutenberg.org/*). Project Gutenberg maintains a collection of literary works that are available in the public domain, and it's a great resource if you're interested in working with literary texts in your programming projects.
- Let's pull in the text of *Alice in Wonderland* and try to count the number of words in the text.
- We'll use the string method split(), which can build a list of words from a string.
- The split() method separates a string into parts wherever it nds a space and stores all the parts of the string in a list.
- The result is a list of words from the string, although some punctuation may also appear with some of the words.
- To count the number of words in *Alice in Wonderland*, we'll use split() on the entire text.
- Then we'll count the items in the list to get a rough idea of the number of words in the text:

Exception Handling

Analyzing Text

```
filename = 'alice.txt'
try:
    with open(filename) as f_obj:
        contents = f_obj.read()
except FileNotFoundError:
    msg = "Sorry, the file " + filename + " does not exist."
    print(msg)
else:
    # Count the approximate number of words in the file.
    words = contents.split()
    num_words = len(words)
    print("The file " + filename + " has about " + str(num_words) + " words.")
```

- If ‘alice.txt’ is in your current working directory, there will be no error and the ouput will be something like :

```
The file alice.txt has about 29461 words.
```

Exception Handling

Analyzing Text – with multiple files

```
def count_words(filename):
 ‘’’ Count the approx no of words in a file’’’
    try:
        with open(filename) as f_obj:
            contents = f_obj.read()
    except FileNotFoundError:
        msg = "Sorry, the file " + filename + " does not exist."
        print(msg)
    else:
        # Count the approximate number of words in the file.
        words = contents.split()
        num_words = len(words)
        print("The file " + filename + " has about " + str(num_words) + " words.")
filenames = [‘alice.txt’, ‘siddhartha.txt’, ‘moby_dick.txt’, ‘little_woman.txt’ ]
for filename in filenames:
    count_words(filename)
```

Exception Handling

Analyzing Text – with multiple files

- Say , if the file ‘siddhartha.txt’ is not in the proper(current) directory, you will get an output something like :

```
The file alice.txt has about 29461 words.
Sorry, the file siddhartha.txt does not exist.
The file moby_dick.txt has about 215136 words.
The file little_women.txt has about 189079 words.
```

Exception Handling

- The try statement lets us capture an exception. When an exception is raised, we have a number of choices for handling it:
- **Ignore it**: If we do nothing, the program stops. We can do this in two ways—don't use a try statement in the first place, or don't have a matching except clause in the try statement.
- **Log it**: We can write a message and let it propagate; generally this will stop the program.
- **Recover from it**: We can write an except clause to do some recovery action to undo the effects of something that was only partially completed in the try clause. We can take this a step further and wrap the try statement in a while statement and keep retrying until it succeeds.

  Silence it: If we do nothing (that is, pass) then processing is resumed after the try statement. This silences the exception.
- **Rewrite it**: We can raise a different exception. The original exception becomes a context for the newly-raised exception.
- **Chain it**: We chain a different exception to the original exception. We'll look at this in the *Chaining exceptions with the raise from statement* recipe.

Exceptions

- Exception handling is done with the try statement.
- The try statement encapsulates several pieces of information.
- Primarily, it contains a suite of statements and a group of exception-handling clauses.
- Each exception-handling clause names a class of exceptions and provides a suite of statements to execute in response to that exception.
- The basic form of a try statement looks like this:

```
try:
    # you do this block
    <suite>
except  :
    # If there is any exception, then execute this block
    <suite>
```

Exceptions

- Each suite is an indented block of statements.
- Any statement is allowed in the suite.
- While this means that you can have nested try statements, that is rarely necessary, since you can have an unlimited number of except clauses on a single try statement.
- If any of the statements in the try suite raise an exception, each of the except clauses are examined to locate a clause that matches the exception raised.
- If no statement in the try suite raises an exception, the except clauses are silently ignored.

- A better way to write this can be :

Exceptions

```
print('Python', python_version())
try:
    let_us_cause_a_NameError
except NameError as err:
    print(err, '--> our error message')
```

Exceptions

- So the general syntax can be :

```
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.
```

Note : Here the else-block is a good place for code that does not need the try

Exceptions

- Let us see an 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()
```

Exceptions

- The try-finally clause:

```
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")
finally :
    print ("Written content in the file successfully")
    fh.close()
```

Exceptions

- The try-except-else-finally clause: -- let us see another example

```
try:
    num1, num2 = eval(input("Enter two numbers, separated by a comma : "))
    result = num1 / num2
    print("Result is", result)
 except ZeroDivisionError:
    print("Division by zero is error !!")
 except SyntaxError:
    print("Comma is missing. Enter numbers separated by comma like this 1, 2")
except:
    print("Wrong input")
else:
    print("No exceptions”)
finally:
    print("This will execute no matter what")
```

Exceptions

- Raising exceptions and printing them
- You can use the following code to assign exception object to a variable.

```
try:
      # this code is expected to throw exception
except ExceptionType as ex:
     # code to handle exception
```

- As you can see you can store exception object in variable ex .
- Now you can use this object in exception handler code

```
try:
     number = eval(input("Enter a number: "))
     print("The number entered is", number)
except NameError as ex:
     print("Exception:", ex)
```

- Here if you run the program and enter numbers, it will run;; but if you enter a string, it will print an error

Reference

- Python for Informatics – C. Severance
- Think Python – A. B. Downey - [O’Reilly]
- Python Crash Course – Eric Matthes [No starch Press]
- A Byte of Python – Swaroop C H
- Introducing Python – Bill Lubanovic [O’Reilly]

- Learning Python – 5th Ed – Mark Lutz - [O’Reilly]

End of Presentation

Python – S9