ECS 32B - Python Concepts for ECS 32B

Aaron Kaloti

UC Davis - Summer Session #2 2020

Overview

Review of Certain ECS 32A Concepts

- Exceptions.
- File I/O.
- User-defined classes.
- References.

Additional Concepts Helpful for ECS 32B

- Unit testing.
- List comprehensions.
- TBA.

Exceptions

Overview

- What they are / how they work. `try`/`except` clauses.
- Raising an exception explicitly: `raise`.
- Exceptions and functions; propagation.
- `finally` clause.

Exceptions

- An **exception** indicates a certain kind of error (with an optional associated message) and forces a program to address the error or crash.

Example #1

```
a = 8
b = 0
print(a / b)
```
foo.py

```
Traceback (most recent call last):
  File "foo.py", line 3, in <module>
    print(a / b)
ZeroDivisionError: division by zero
```

- The line `print(a / b)` (specifically the operation `a / b`) raised/threw[1] a `ZeroDivisionError` exception.

Example #2

```
a = b - 4
```
goo.py

```
Traceback (most recent call last):
  File "goo.py", line 1, in <module>
    a = b - 4
NameError: name 'b' is not defined
```

- The line `a = b - 4` raised/threw a `NameError` exception.

1. "raise" is the Python term. In C/C++, the term "throw" is used instead.

Catching Exceptions

- Usually, when an exception is thrown, the program crashes.
- Use `try`/`except` clauses to prevent this.
  - `try` clause: try to execute all of the code in this block. If successful, then ignore the `except` clause.
  - `except` clause: *immediately* go this clause the moment an exception is raised/thrown in the `try` clause.

Example #1

```
try:
    print("AAA")
    a = 5 / 0
    print("BBB")
except:
    print("You divided by zero, dummy!")
```

my-first-exception.py

- Output:

```
AAA
You divided by zero, dummy!
```

Catching Exceptions

- Avoids a crash, so lines after the `try`/`except` clauses can be executed.

Example #2

```
try:
    val = int(input("Enter: "))
    print("You entered:", val)
except:
    print("You did not enter an integer.")

print("This will always be printed.")
```

my-second-exception.py

Executions

```
Enter: 5
You entered: 5
This will always be printed.
```

```
Enter:     5
You entered: 5
This will always be printed.
```

```
Enter: 5a
You did not enter an integer.
This will always be printed.
```

Catching Exceptions

- Exceptions not thrown in a `try` block won't be caught.

Example

```
a = 5 / 0
try:
    print(a)
except:
    print("Dividing by zero is so not cool!")

print("This won't get printed.")
```

raised-too-early.py

Execution

```
Traceback (most recent call last):
  File "raised-too-early.py", line 1, in <module>
    a = 5 / 0
ZeroDivisionError: division by zero
```

Raising Exceptions Explicitly

- Can explicitly raise exception with `raise` keyword.
  - The specific exception that you raise (`ZeroDivisionError`, `NameError`, `ValueError`, etc.) doesn't have to make sense, but you should try to use an appropriate one.
- Useful when writing code for other people to use.
  - This will make more sense once we get to user-defined classes.

Example #1

```
print("This program will intentionally crash right about now.")
raise ValueError
print("This line won't be reached.")
```

raise0.py

- Output:

```
This program will intentionally crash right about now.
Traceback (most recent call last):
  File "raise0.py", line 2, in <module>
    raise ValueError
ValueError
```

Raising Exceptions Explicitly

Example #2

```
try:
    print("AAA")
    raise ZeroDivisionError
    print("BBB")
except:
    print("CCC")
print("DDD")
```

raise1.py

- Output:

```
AAA
CCC
DDD
```

Raising Exceptions Explicitly

- Can attach message to the exception and then raise it.
  - Again, although the message needn't be relevant, you should try to make it relevant.

Example #3

```
print("This program will intentionally crash right about now.")
raise FileNotFoundError("Fish tacos!")
print("This line won't be reached.")
```

raise2.py

- Output:

```
This program will intentionally crash right about now.
Traceback (most recent call last):
  File "raise2.py", line 2, in <module>
    raise FileNotFoundError("Fish tacos!")
FileNotFoundError: Fish tacos!
```

Propagation

- An exception not caught in a function will *propagate* outside of the function, where it still can be caught. That is, the function that throws the exception does not need to be the one to catch it.

Example #1

```
def foo(a, b):
    c = a / b
    return c

try:
    print("AAA")
    foo(2, 0)
    print("BBB")
except:
    print("Caught exception.")
```

- Output:

```
AAA
Caught exception.
```

Propagation

- If an exception is raised *and* caught within the same function, it won't propagate outside of the function. The caller will never know or be able to check that an exception was raised/caught in the function it called.

Example #2

```
def foo(a, b):
    try:
        c = a / b
        return c
    except:
        return -1

try:
    print("AAA")
    foo(2, 0)
    print("BBB")
except:
    print("Caught exception.")
```

```
AAA
BBB
```

Catching Specific Exceptions

Example #1

```
try:
    a = int(input("Enter: "))
    b = int(input("Enter: "))
    c = a / b
    foo()      # nonexistent function
except ZeroDivisionError:
    print("You divided by zero!")
except ValueError:
    print("You did not enter a valid integer!")
except:
    print("Something truly bizarre happened!")
```

catching-specific-exceptions.py

Executions

```
Enter: 3
Enter: 0
You divided by zero!
```

```
Enter: 3a
You did not enter a valid integer!
```

```
Enter: 3
Enter: 5
Something truly bizarre happened!
```

Catching Specific Exceptions

Example #2

```
def bar():
    try:
        f = open("blajasjfklasjl")
    except:
        print("EXCEPTION")
        raise ValueError

try:
    a = int(input("Enter: "))
    b = int(input("Enter: "))
    c = a / b
    bar()
except ZeroDivisionError:
    print("You divided by zero!")
except ValueError:
    print("You did not enter a valid integer!")
except:
    print("Something truly bizarre happened!")
```

catching-specific-exceptions2.py

Execution

```
Enter: 3
Enter: 5
EXCEPTION
You did not enter a valid integer!
```

Remarks

- If the `raise ValueError` were removed, then the `except ValueError` exception handler would not have been triggered, because no exception would *propagate* out of `bar()`.

Raising/Throwing Exceptions

- When you `raise` an exception, you can specify the message associated with it. Compare the three scenarios shown below.

```
>>> a = 5 / 0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a = 5 / 0
ZeroDivisionError: division by zero
>>> raise ZeroDivisionError
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    raise ZeroDivisionError
ZeroDivisionError
>>> raise ZeroDivisionError("iowjwiojioajls")
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    raise ZeroDivisionError("iowjwiojioajls")
ZeroDivisionError: iowjwiojioajls
```

`finally` Clause

- Can specify something that should *always* happen after a corresponding `try`/`except` setup ends.

Example

```
try:
    divisor = int(input("Enter divisor: "))
    result = 5 / divisor
    print("5 divided by {} is {}".format(divisor, result))
except ZeroDivisionError:
    print("Dividing by zero is bad!")
finally:
    print("Good effort!")
```

```
Enter divisor: 13
5 divided by 13 is 0.38461538461538464
Good effort!
```

```
Enter divisor: 0
Dividing by zero is bad!
Good effort!
```

```
Enter divisor: blah
Good effort!
Traceback (most recent call last):
  File "finally1.py", line 2, in <module>
    divisor = int(input("Enter divisor: "))
ValueError: invalid literal for int() with base 10: 'blah'
```

File I/O

Overview

- Opening files.
- Closing files.
- Reading from files.
- Writing to files.
- Appending to files.
- `with` statement.

File I/O: Basics

Example: `copy_matches()`

Prompt

- Write a function called `copy_matches()` that takes three strings arguments. The first and third arguments are file names, with the first file name referring to the input file. `copy_matches()` should copy each line in this text file that contains the string given by the second argument to a file whose name is given by the third argument. Put informally, this function copies all lines from the input file to the output file that contain the target string. The original contents of the output file should be erased.

Solution

- *After lecture, the solution will be pasted into the slide here (or onto the next slide if it cannot fit on this one).*

Appending to a File

Example: `append_matches()`

Prompt

- Modify `copy_matches()` (call the modified version `append_matches()`) so that it does not erase the original contents of the output file.

Solution

- *After lecture, the solution will be pasted into the slide here (or onto the next slide if it cannot fit on this one).*

Iterating through a File

Examples

```
Line 1
Line 2
Line 3
```

testfile.txt

```
f = open("testfile.txt")
print("read():")
print(f.read(), end='')
f.seek(0)  # back to start
print("readline():")
line = f.readline()
while line != "":
    print(line, end='')
    line = f.readline()
f.seek(0)
print("readlines():")
lines = f.readlines()
for line in lines:
    print(line, end='')
f.seek(0)
print("for ... in file:")
for line in f:
    print(line, end='')
f.close()
```

file_demo.py

```
read():
Line 1
Line 2
Line 3
readline():
Line 1
Line 2
Line 3
readlines():
Line 1
Line 2
Line 3
for ... in file:
Line 1
Line 2
Line 3
```

`with` Statement

- Can open a file that'll automatically close once exit `with` clause, regardless of how exited (whether naturally, due an exception, due to returning, etc.).

```
with open("testfile.txt") as f:
    print(f.read(), end='')
```
with_demo.py

```
Line 1
Line 2
Line 3
```
testfile.txt

- Can use with multiple files.

```
with open("file1.txt") as readfile, open("file2.txt") as readfile2:
    print("File #1 closed 1: {}".format(readfile.closed))
    print("File #2 closed 1: {}".format(readfile2.closed))

print("File #1 closed 2: {}".format(readfile.closed))
print("File #2 closed 2: {}".format(readfile2.closed))
```

```
File #1 closed 1: False
File #2 closed 1: False
File #1 closed 2: True
File #2 closed 2: True
```

User-Defined Classes

Definition

- **class**: blueprint for how a new type (that we define) operates.
  - a.k.a. **programmer-defined type**, **user-defined class**.
  - Lets us tell our code about real-world concepts that Python doesn't know about.

User-Defined Classes

Example: `Point` Class

```
from math import sqrt

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({},{})".format(self.x,self.y)

    # Compute distance from origin.
    def distance(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def distance_between(self,other):
        x_dist2 = (self.x - other.x) ** 2
        y_dist2 = (self.y - other.y) ** 2
        return sqrt(x_dist2 + y_dist2)
```

point.py

```
from point import Point

p1 = Point(3, -4)
p2 = Point(6, 8)
print(p1)
print(p2)
print(p1.distance())
print(p2.distance())
print(p1.distance_between(p2))
print(p2.distance_between(p1))
print(p1.distance_between(p1))
```

try_point.py

```
(3,-4)
(6,8)
5.0
10.0
12.36931687685298
12.36931687685298
0.0
```

Terminology

- `p1` and `p2` are generically called *objects*. (Lists and dictionaries are also objects.)
- `p1` and `p2` are **instances** of `class Point`.
- `x` and `y` are class members / data members / fields of `p1` and `p2`.
- `p1 = Point(...)` and `p2 = Point(...)` are acts of **instantiation**.
- `__init__()` is the initializer.
- `__init__()`, `__str__()`, `distance()`, and `distance_between()` are methods.

User-Defined Classes

Example: Soda Machine

Prompt

- Implement two classes, `Soda` and `SodaMachine`. `Soda` just has a name and a size. `SodaMachine` should have a member to keep track of the amount of money and methods to allow the user to add quarters and try to get a soda.

Solution

- *After lecture, the solution will be pasted into the slide here (or onto the next slide if it cannot fit on this one).*

References[1]

1. I have to make more slides for this than I had in ECS 32A, since it was not a significantly stressed concept, at least in my ECS 32A.

References / Further Reading

- Textbook I used as primary reference for ECS 32A: *Think Python: How to Think Like a Computer Scientist* by Allen Downey.
  - Freely available here.
  - Potentially more navigable website version here; the content is the same, but it is presented as webpages instead of as a PDF.