CSC148 Week 3

Larry Zhang

Announcements

- Exercise 1 is out, due January 26
- Individual submission on MarkUs, make sure you can login, if not let us know.

Exceptions

Behaviours of Python functions

So far, our Python functions have returned values or modified objects

if everything goes well and the function can complete **successfully** …

What if the function **cannot** complete successfully?

- calling `s.pop()` on an empty stack `s`
- evaluating `x = 3 / 0`

In general, we want the program to be able to express something like "**there is an error of some type and I'm not completing this function successfully!**"

Also, we don't want our program to **crash** all the time, we want to **handle** the error whenever possible.

Approach #1: return error code

- In some languages (e.g., C), functions **return** “special values” to signify errors.
  - e.g., UNIX functions return -1 to mean error

DEMO:

`no_error_handling.py` → `return_error_code.py`

```
def f1():

    x = f2()
    return x * 2 - 11

def f2():

    x = f3()
    return x + 1

def f3():

    d = random.randint(0, 5)
    print("d =", d)
    return 20 // d

if __name__ == "__main__":

    res = f1()
    print("The result is", res)
```

```
def f1():

    x = f2()
    if x == -1:      # if f2() had an error
        return -1    # return error
    return x * 2 - 11

def f2():

    x = f3()
    if x == -1:      # if f3() had an error
        return -1    # return error
    return x + 1

def f3():

    d = random.randint(0, 5)  # what if d = 5 ??
    print("d =", d)
    if d == 0:
        return -1    # return error code -1
    return 20 // d

if __name__ == "__main__":

    res = f1()
    if res == -1:
        print("There is an error!")
    else:
        print("The result is", res)
```

Approach #1: return error code

- In some languages (e.g., C), functions **return** “special values” to signify errors.
  - e.g., UNIX functions return -1 to mean error
- **Concerns** with this approach:
  - It requires you to check the return value of every function you call.
    - this changes the flow of the program
  - It assumes there is an appropriate error value to return that won’t be confused with a real return value!
    - in `return_error_code.py`, when the randomly generated d = 5, the correct return value is -1, which is confused with the error code!

Approach #2: Use **Exceptions**

An **exception** is an **object** that indicates an exceptional situation (not necessarily a problem)

An exception can be **raised** during program execution, and when it is raised the control of the program is transferred to some **exception handler**.

Exceptions must be "**caught**" by an exception handler somewhere along the line, or your program will crash.

Exceptions allow you to structure code in a natural way so that error handling and recovery are **isolated** from the regular flow of your program. (We will see why).

DEMO
some types of exceptions

```
>>> 10 * (1 / 0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + junk
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'junk' is not defined
>>> junk = 'abc'
>>> 4 + junk # can't add int and str
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: ...
```

DEMO
raise some exceptions

Two forms:

```
>>> raise ValueError
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError
>>> raise ValueError('invalid time/date value')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid time/date value
```

User-Defined Exceptions

You can make new types of exceptions by ...

inheriting from the **Exception** class

```
>>> class ExtremeException(Exception):
...   pass
...
>>> raise ExtremeException('unrecoverable error')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
__main__.ExtremeException: unrecoverable error
```

Handling Exceptions

Handling Exceptions

If a piece of code can raise an exception, you can put it in a **`try`** block.

You can have multiple **`except`** blocks that handle different types of exceptions

An **`except`** block "matches" a raised exception if the **`except`** block names the **same class** or **a superclass** of the exception

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s)
    print('success')
except IOError:
    print('Input/Output error.')
except ValueError:
    print('Could not convert data

print('continuing')
```

This catches all possible exceptions, but it's **BAD** practice. Don't do it!

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s)
    print('success')
except Exception:
    print('some error')

print('continuing')
```

Execution with Exceptions

If no exception occurs, no except block is executed

If an exception occurs and an except block matches it, run the handling code in the except block, then execution continues following the enclosing try-except block.

If an exception occurs and no except block handles it, the exception propagates up the function call stack until it is handled or terminates the program.

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s)
    print('success')
except IOError:
    print('Input/Output error.')
except ValueError:
    print('Could not convert data

print('continuing')
```

DEMO
Example of handling exception
`e1.py`

DEMO

`no_error_handling.py` → `use_exception.py`

returning error code VS using exception

```python
def f1():

    x = f2()
    if x == -1:      # if f2() had an error
        return -1    # return error
    return x * 2 - 11

def f2():

    x = f3()
    if x == -1:      # if f3() had an error
        return -1    # return error
    return x + 1

def f3():

    d = random.randint(0, 5)  # what if d = 5 ??
    print("d =", d)
    if d == 0:
        return -1    # return error code -1
    return 20 // d

if __name__ == "__main__":

    res = f1()
    if res == -1:
        print("There is an error!")
    else:
        print("The result is", res)
```

VS

```python
def f1():

    x = f2()
    return x * 2 - 11

def f2():

    x = f3()
    return x + 1

def f3():

    d = random.randint(0, 5)
    print('d =', d)
    return 20 // d

if __name__ == '__main__':

    try:
        res = f1()
        print('The result is', res)
    except ZeroDivisionError:
        print('There is an error')
```

Error handling is **isolated** from the regular program flow.

Avoid the issue when -1 is a legit return value.

Tracing Exception Code (e2.py)

```
def bad():
  return 2 / 0

def bad_caller():
  try:
    bad()
  except ValueError:
    print("Caught exception in bad_caller!")

def entry():
  try:
    bad_caller()
  except ZeroDivisionError:
    print("Caught exception in entry!")

entry()
```

Tracing Exception Code, again (e3.py)

```
def bad():
  return 2 / 0

def bad_caller():
  try:
    bad()
  except ArithmeticError:
    print("Caught exception in bad_caller!")

def entry():
  try:
    bad_caller()
  except ZeroDivisionError:
    print("Caught exception in entry!")

entry()
```

Hint:
`ArithmeticError` is a superclass of `ZeroDivisionError`

DEMO
Tracking Exception Code
`e2.py` and `e3.py`

Finally

A **try** block may also have a **finally** block

Code in the finally block runs regardless of whether there is an exception and regardless of whether that exception is caught.

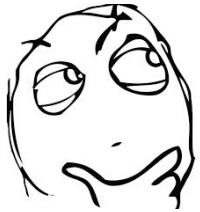
It can be used for cleanup actions that must always occur.

DEMO: `finally.py`

```
ok = False
try:
    f = open('myfile.txt')
    ok = True
except IOError:
    print('Input/Output error.')

if ok: # if True, file must be open
    try:
        s = f.readline()
        i = int(s)
    except ValueError:
        print('Could not convert data
    finally:
        f.close()
        print('File closed')
```

Why bother having **`finally`**?



We could just put clean-up actions outside after the **try-except** block, right?

```
try:
    a = 1
    raise ValueError("not caught!")
except TypeError:
    print("type error")
finally:
    print("clean up")
```

“clean up” is printed before crashing

```
try:
    a = 1
    raise ValueError("not caught!")
except TypeError:
    print("type error")

print("clean up")
```

it crashes and “clean up” is NOT printed

Else

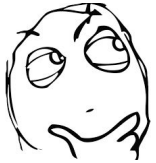
A **try** block may also have an **else** block.

Code in the else block runs exactly if **no exceptions** are raised in the **try** code.

DEMO: `else.py`

```
try:
    print('hello')
except ZeroDivisionError:
    print('caught')
else:
    print('else')
finally:
    print('finally')
```

Why bother having `else`?



Why not just put the lines in the "`else`" block in the "`try`" block, or outside the try-except block?

Example: Suppose we we want the following:

- open `file2` **only if** `file1` has been opened successfully
- if `file2` does not exist, let it crash rather than catching the exception

```
try:
    file1 = open("file1.txt")
    print("file1 opened")
except IOError:
    print("not crashing")
else:
    file2 = open("file2.txt")
    print("file2 opened")
```

```
try:
    file1 = open("file1.txt")
    print("file1 opened")
    file2 = open("file2.txt")
    print("file2 opened")
except IOError:
    print("not crashing")
```

```
try:
    file1 = open("file1.txt")
    print("file1 opened")
except IOError:
    print("not crashing")

file2 = open("file2.txt")
print("file2 opened")
```

Python Idioms

read idioms.pdf