Computational Structures in Data Science

Lecture:
Exceptions

Week 6, Summer 2024. 7/23 (Tues)

Announcements

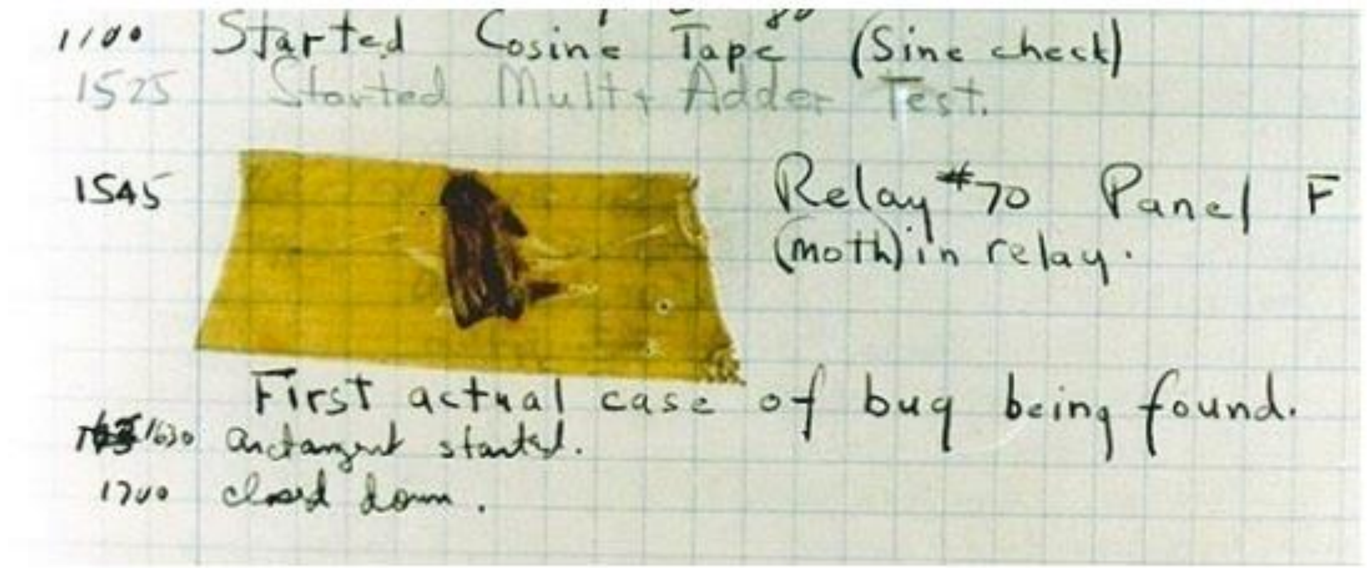
- Project02 (“Ants”) out!
- Mid-point course survey out (optional, +2 extra credit points)
  - On Gradescope, named “Extra Credit Mid-semester Feedback”
  - Due: 7/29 11:59 PM PST

Learning Objectives

- Exceptions give us a formal way to address error conditions
- "Catch" exceptions in a Python Program
- Define and Raise our own exceptions

Errors Can Occur Just About Anywhere!

- Function receives arguments of improper type?
- Resources (e.g. files or some data) are not available
- Network connection is lost or times out?



1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

~~1545~~ 1630 Antangent started.

1700 closed down.

Grace Hopper's Notebook, 1947, Moth found in a Mark II Computer

Error Handling Approach: int status code

- Historically, there are simple (yet primitive) error handling approaches
- int error codes: Each function returns (or sets) a status code to indicate whether a call failed/succeeded.

```python
# Define int status codes
STATUS_CODE_SUCCESS = 0
STATUS_CODE_VIDEO_URL_INVALID = 1
STATUS_CODE_DOWNLOAD_FAILURE = 2
STATUS_CODE_DOWNLOAD_THROTTLED = 3
STATUS_CODE_VIDEO_IS_PRIVATE = 4
# ...many more...

def fetch_video(video_url):
    if not is_valid_url(video_url):
        return STATUS_CODE_VIDEO_URL_INVALID, None
    download_status, video = download_video(video_url)
    if download_status == STATUS_CODE_DOWNLOAD_FAILURE:
        return STATUS_CODE_DOWNLOAD_FAILURE, None
    elif download_status == STATUS_CODE_DOWNLOAD_THROTTLED:
        return STATUS_CODE_DOWNLOAD_THROTTLED, None
    if is_private(video):
        return STATUS_CODE_VIDEO_IS_PRIVATE, None
    return STATUS_CODE_SUCCESS, video
```

Error Handling Approach: Exceptions

- Idea: add a new language feature for error handling
- Called “Exceptions”
- Pros
  - Exceptions keep track of where in the code the error came from (eg function name, line number). Very useful for debugging
- Cons
  - Requires language support. Implementation is nontrivial

```python
def fetch_video(video_url):
    if not is_valid_url(video_url):
        raise InvalidUrl(video_url)
    video = download_video(video_url)
    if is_private(video):
        raise VideoIsPrivate(video_url)
    return video
```

`download_video()` may throw its own exceptions like “VideoNotFound” or “DownloadThrottled”

New Python syntax: `raise`! Syntax for raising (aka throwing) an Exception

Raise statement

- Exception are raised with a raise statement
  - raise <exception>, e.g.:
  - `raise NameError(f"The property {name} does not exist")`

- <expression> must evaluate to a subclass of BaseException or an instance of one
- Exceptions are constructed like any other object
  - TypeError('Bad argument')
- Raise Exceptions for unrecoverable errors!
  - Something bad has gone on and you cannot continue.

What does an Exception look like? Example exceptions (Docs)

- Unhandled, "thrown" back to the top level interpreter
- Or halt the program

```
>>> 3/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> str.lower(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'lower' requires a 'str' object
but received a 'int'
>>> ""[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

Exceptions contain useful info:

(1) Exception name/message. Tells you what went wrong.

(2) Error location. Tells you exactly where in the code the Exception was thrown (`raise`).

What does an Exception look like? Example exceptions (Docs)

```python
def square_sum_nums(lst_nums):
    if not isinstance(lst_nums, list):
        raise TypeError(f"Expected list, got: {type(lst_nums)}")
    return sum(map(lambda num: num ** 2, lst_nums))

def fn_a():
    my_var = 42
    x = square_sum_nums(my_var)
    return x + 2
```

```
>>> fn_a()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".\lecture19.py", line 36, in fn_a
    x = square_sum_nums(my_var)
  File ".\lecture19.py", line 31, in square_sum_nums
    raise TypeError(f"Expected list, but got: {type(lst_nums)}")
TypeError: Expected list, but got: <class 'int'>
```

Exceptions contain useful info:

(1) Exception name/message. Tells you what went wrong.

(2) Error location. Tells you exactly where in the code the Exception was thrown (`raise`).

Reading A "Stack Trace" or "Traceback" (Docs)

- All errors in Python *should* return some structured feedback.
- Errors may be dense but contain some really helpful information!

```
👉 python3 -i 18-Exceptions.py
What is your age? 5
Catching CS88Error
Traceback (most recent call last):
  File "…Exceptions.py", line 24, in <module>
    get_age_in_days()
  File "…", line 20, in get_age_in_days
    raise e
  File "…", line 14, in get_age_in_days
    raise CS88Error('You seem young!')
__main__.CS88Error: You seem young!
```

Pro Tip: learning how to read an error stack trace is an extremely useful skill that will help you in this class, your other classes, and your future career.

(aka don't just "eye-glaze" when faced with a stack trace)

What do Exceptions do? Exceptional exit from functions

- Function doesn’t “return” but instead execution is thrown out of the function

```
>>> def divides(x, y):
...     return y % x == 0
...
>>> divides(0, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in divides
ZeroDivisionError: integer division or modulo by zero
>>> def get(data, selector):
...     return data[selector]
...
>>> get({'a': 34, 'cat':'9 lives'}, 'dog')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in get
KeyError: 'dog'
>>>
```

Continue out of multiple calls deep

- Stack “unwinds” until exception is **handled** or we reach the top level frame (program crash!)

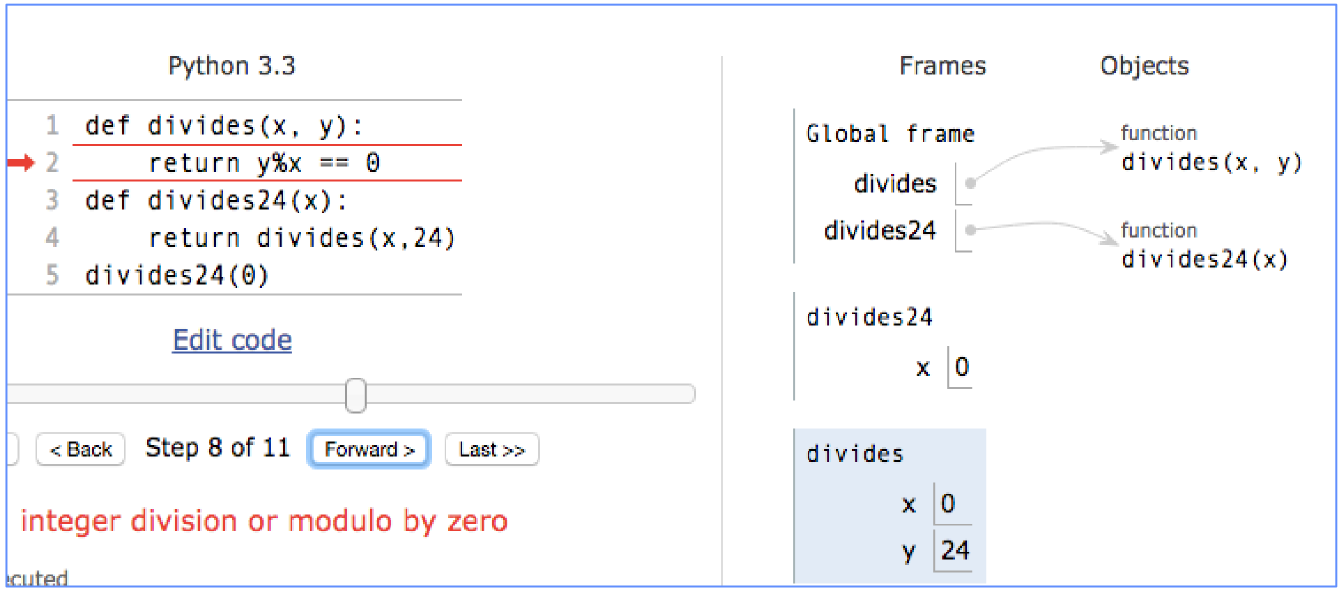
```python
def divides(x, y):
    return y%x == 0
def divides24(x):
    return divides(x,24)
divides24(0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-14-ad26ce8ae76a> in <module>()
      3 def divides24(x):
      4     return divides(x,24)
----> 5 divides24(0)

<ipython-input-14-ad26ce8ae76a> in divides24(x)
      2     return y%x == 0
      3 def divides24(x):
----> 4     return divides(x,24)
      5 divides24(0)

<ipython-input-14-ad26ce8ae76a> in divides(x, y)
      1 def divides(x, y):
----> 2     return y%x == 0
      3 def divides24(x):
      4     return divides(x,24)
      5 divides24(0)

ZeroDivisionError: integer division or modulo by zero
```



```python
def divides(x, y):
    return y%x == 0
def divides24(x):
    return divides(x,24)
divides24(0)
```

integer division or modulo by zero

```
Global frame
    divides      function divides(x, y)
    divides24    function divides24(x)

divides24
    x  0

divides
    x  0
    y  24
```

Flow of control stops at the exception

- And is 'thrown back' to wherever it is caught, by default no where.

```
def throws_exception():
    raise Exception("lol")
    print("Never gets here")
    return 42
```

This print statement never happens, as the Exception interrupts the execution and "jumps" out of the function back to the caller

```
>>> x = throws_exception()
Traceback (most recent call last):
  File ".\lecture19.py", line 44, in <module>
    x = throws_exception()
  File ".\lecture19.py", line 41, in throws_exception
    raise Exception("lol")
Exception: lol
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Important: `x` is never assigned to! The Exception interrupted the assignment statement.
A sign that Exceptions in Python are indeed a Special Thing

Types of exceptions

- Exceptions are just classes in Python, with common types for ease of use / clarity.
  - All inherit from `BaseException`
- `AssertionError` – The of exception raised by a failing assert statement
- `TypeError` -- A function was passed the wrong number/type of argument
- `NameError` -- A name wasn't found
- `KeyError` -- A key wasn't found in a dictionary
- `RuntimeError` -- Catch-all for troubles during interpretation
- Your own exceptions!

Assert Statements

- Allow you to make assertions about assumptions that your code relies on
  - Use them liberally!
  - Incoming data is "dirty" and unsafe till you've "cleaned" it

```
assert <assertion expression>, <string for failed>
```

- They "do nothing" if the statement is true.
- Raise an exception of type `AssertionError`
- You can turn them off:
  - Ignored in optimize flag: `python3 -O` ...
  - Governed by bool __debug__

```
def divides(x, y):
    assert x != 0, ”Denominator
must be non-zero”
    return y % x == 0
```

Handling Exceptions

- What if I wanted to try to recover from an Exception?
- Try/except to the rescue!

```python
def fetch_video_recover(video_url):
    if not is_valid_url(video_url):
        raise InvalidUrl(video_url)
    # Idea: if video download fails,
    # wait 2 seconds and retry...
    video = download_video(video_url)
    if is_private(video):
        raise VideoIsPrivate(video_url)
    return video
```

```python
def fetch_video_recover(video_url):
    if not is_valid_url(video_url):
        raise InvalidUrl(video_url)
    try:
        video = download_video(video_url)
    except DownloadFailed:
        # wait for 2 seconds, and try again
        # if this 2nd try fails, then fail
        # the entire function call
        import time
        time.sleep(2)  # wait for 2 seconds
        video = download_video(video_url)
    if is_private(video):
        raise VideoIsPrivate(video_url)
    return video
```

New Python syntax: `try`, `except`

Handling Errors – `try / except`

- Wrap your code in try – except statements

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
... # continue here if <try suite> succeeds w/o exception
```

- Execution rule
  - <try suite> is executed first
  - If during this an exception is raised and not handled otherwise
  - And if the exception inherits from <exception class>
  - Then <except suite> is executed with <name> bound to the exception
- Control jumps to the except suite of the most recent try that handles the exception

Demo: safe_apply_fn

```python
def safe_apply_fn(f, x, y):
    try:
        return f(x, y)
    except Exception as e:
        return e

def divides(x, y):
    assert x != 0, "Bad arg to divides, denom must be non-zero"
    if (type(x) != int or type(y) != int):
        raise TypeError("divides only takes integers")
    return y % x == 0
```

```
>>> safe_apply_fn(divides, 3, 9)
True
>>> safe_apply_fn(divides, 0, 9)
Bad arg to divides, denom must be non-zero
>>> safe_apply_fn(divides, 'hi', 42)
divides only takes integers
```

```
>>> divides(0, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".\lecture19.py", line 82, in divides
    assert x != 0, "Bad arg to divides,
denom must be non-zero"
AssertionError: Bad arg to divides, denom
must be non-zero
```

Exceptions are Classes

```
class NoiseyException(Exception):
    def __init__(self, stuff):
        print("Bad stuff happened", stuff)
```

```
class CS88Error(Exception):
    pass # The one time you can skip init. ;)
```

```
try:
    return fun(x)
except:
    raise NoiseyException((fun, x))
```

Summary

- Approach use of exceptions as a design problem
  - Meaningful behavior => methods [& attributes]
  - ADT methodology: What should a function do?
  - What’s private and hidden? vs What’s public?
- Use it to streamline development

- Anticipate exceptional cases and unforeseen problems
  - try … except
  - raise / assert