UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Lecture #12: Exceptions and Iterators

November 25, 2019 http://inst.eecs.berkeley.edu/~cs88

1

Computational Concepts Toolbox

- Data type: values, literals, operations,
- Expressions, Call expression
- Variables
- Assignment Statement, Tuple assignment
- Sequences: tuple, list
- Dictionaries
- Function Definition Statement
- Conditional Statement
- Iteration: list comp, for, while
- Lambda function expr.
- Higher Order Functions
  - Functions as Values
  - Functions with functions as argument
  - Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions
- Recursion
- Abstract Data Types
- Mutation
- Class & Inheritance
- Exceptions
- Iterators & Generators

2

Today:

- Exceptions
- Sequences vs Iterables
- Using iterators without generating all the data
- Magic methods
  - `next`
  - `Iter`
- Iterators – the iter protocol
- Getitem protocol
- Is an object iterable?
- Lazy evaluation with iterators

3

Exception (read 3.3)

- Mechanism in a programming language to declare and respond to "exceptional conditions"
  - enable non-local cntinuations of control
- Often used to handle error conditions
  - Unhandled exceptions will cause python to halt and print a stack trace
  - You already saw a non-error exception – end of iterator
- Exceptions can be handled by the program instead
  - `assert, try, except, raise` statements
- Exceptions are objects!
  - They have classes with constructors

4

Handling Errors

- Function receives arguments of improper type?
- Resource, e.g., file, is not available
- Network connection is lost or times out?

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

5

Example exceptions

notebook

```
>>> 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
>>>
```

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

6

Functions

- Q: What is a function supposed to do?
- A: One thing well
- Q: What should it do when it is passed arguments that don't make sense?

```
>>> def divides(x, y):
...     return y%x == 0
...
>>> divides(0, 5)
???

>>> def get(data, selector):
...     return data[selector]
...
>>> get({'a': 34, 'cat':'9 lives'}, 'dog')

????
```


7

Exceptional exit from functions

```
>>> 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'
>>>
```

- Function doesn't "return" but instead execution is thrown out of the function

8

Continue out of multiple calls deep

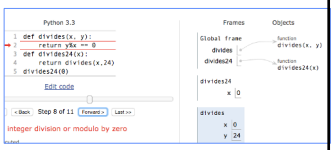
```
def divides(x, y):
    return y%x == 0
def divides24(x):
    return divides(x,24)
divides24(0)
```

```
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
```



- "Stack" unwinds until exception is handled or top

9

Types of exceptions

- `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
- . . .

10

Demo

11

Flow of control stops at the exception

- And is 'thrown back' to wherever it is caught

```
def divides24(x):
    return noisy_divides(x,24)
```

```
divides24(0)
```

```
ZeroDivisionError                         Traceback (most recen
<ipython-input-24-ea94e81be222> in <module>()
----> 1 divides24(0)

<ipython-input-23-c56bc11b3032> in divides24(x)
      1 def divides24(x):
----> 2     return noisy_divides(x,24)

<ipython-input-20-df96adb0c18a> in noisy_divides(x, y)
      1 def noisy_divides(x, y):
----> 2     result = (y % x == 0)
      3     if result:
      4         print("{0} divides {1}".format(x, y))
      5     else:

ZeroDivisionError: integer division or modulo by zero
```


12

Assert Statements

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

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

- Raise an exception of type `AssertionError`
- 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
```


13

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

14

Demo

```
def safe_apply_fun(f,x):
    try:
        return f(x)          # normal execution, return the result
    except Exception as e:   # exceptions are objects of class deri
        return e             # value returned on exception
```

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


15

Raise statement

- Exception are raised with a `raise` statement\

```
raise <exception>
```

- <expression> must evaluate to a subclass of BaseException or an instance of one
- Exceptions are constructed like any other object

```
TypeError('Bad argument')
```


16

Exceptions

What does Python Display?

>>> my_list = ['We', 'Beat', '$tanford']
>>> my_list[3]

A) "GO BEARS!"
B) "$tanford"
C) None
D) IndexError

Solution:
A) An object is an instance of a class

17

Exceptions are Classes

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

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


18

Demo

19

Mind Refresher 2

A setter method…

A) constructs an object
B) changes the internal state of an object or class
C) is required by Python to access variables
D) All of the above

Solution:
B) Changes the internal state of an object or class by allowing access to a private variable.

20

Types of exceptions

- **`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**
- **. . .**

21

Mind Refresher 3

Exceptions…

A) allow to handle errors non-locally
B) are objects
C) cannot happen within a catch block
D) B, C
E) A, B

Solution:
B, C) Exceptions are objects and they can occur any time.

22

Iterable - an object you can iterate over

- *iterable*: An object capable of yielding its members one at a time.
- *iterator*: An object representing a stream of data.
- We have worked with many iterables as if they were sequences
- A Sequence is an iteratable that knows its length and can get a specific item (e.g. a List)

23

Functions that return iterables

- map
- range
- zip

- These objects are not sequences.
- If we want to see all the elements at once, we need to explicitly call list() or tuple() on them

24

Next element in generator iterable

- Iterables work because they have some "magic methods" on them. We saw magic methods when we learned about classes,
- e.g., __init__, __repr__ and __str__.
- The first one we see for iterables is __next__

- `iter( )` – transforms a sequence into an iterator

25

Iterators – iter protocol

- In order to be *iterable*, a class must implement the **iter protocol**
- The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:
  - __iter__() : Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements.
  - This method returns an iterator object, Iterator can be self
  - __next__() : Return the next item from the container. If there are no further items, raise the StopIteration exception.
- Classes get to define how they are iterated over by defining these methods

26

Getitem protocol

- Another way an object can behave like a sequence is *indexing*: Using square brackets "[ ]" to access specific items in an object.
- Defined by special method: **`__getitem__`**`(self, i)`
  - Method returns the item at a given index

```
class myrange2:
    def __init__(self, n):
        self.n = n

    def __getitem__(self, i):
        if i >= 0 and i < self.n:
            return i
        else:
            raise IndexError

    def __len__(self):
        return self.n
```


27

Demo

28

What would Python Display?

```
>>> my_list = ['We, 'Beat', '$tanford']
>>> my_iter = iter(my_list)
>>> next(my_iter)
>>> next(my_iter)
>>> next(my_iter)
>>> next(my_iter)
```

A) "$tanford"
B) None
C) IndexError
D) StopIterationError

Solution: D) iterators stop with a StopIterationError.

29

Determining if an object is iterable

- `from collections.abc import Iterable`
- `isinstance([1,2,3], Iterable)`

- This is more general than checking for any list of particular type, e.g., list, tuple, string...

30