



Computational Structures in Data Science



UC Berkeley EECS
Adj. Ass. Prof.
Dr. Gerald Friedland

Iterators and Generators



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





Today:

- **Review Exceptions**
- **Sequences vs Iterables**
- **Using iterators without generating all the data**
- **Generator concept**
 - Generating an iterator from iteration with `yield`
- **Magic methods**
 - `next`
 - `Iter`
- **Iterators – the `iter` protocol**
- **Getitem protocol**
- **Is an object iterable?**
- **Lazy evaluation with iterators**



Key concepts to take forward

- Classes embody and allow enforcement of ADT methodology
- Class definition
- Class namespace
- Methods
- Instance attributes (fields)
- Class attributes
- Inheritance
- Superclass reference



Summary of last week

- **Approach creation of a class as a design problem**
 - Meaningful behavior => methods [& attributes]
 - ADT methodology
 - What's private and hidden? vs What's public?
- **Design for inheritance**
 - Clean general case as foundation for specialized subclasses
- **Use it to streamline development**

- **Anticipate exceptional cases and unforeseen problems**
 - try ... catch
 - raise / assert



Mind Refresher 1

An object is...

- A) an instance of a class
- B) a python thing
- C) inherited from a class
- D) All of the above



Solution:

- A) An object is an instance of a class



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.**



Exception (read 3.3)

- Mechanism in a programming language to declare and respond to “exceptional conditions”
 - enable non-local continuations 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



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



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
- . . .



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 derived
        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
```



Exceptions are Classes

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

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



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:

- A, B) Exceptions are objects, passed through the callstack and they can occur any time.



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



Functions that return iterables

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



Generators: turning iteration into an iterable

- Generator functions use iteration (for loops, while loops) and the yield keyword
- Generator functions have no return statement, but they don't return None
- They implicitly return a generator object
- Generator objects are just iterators

```
def squares(n):  
    for i in range(n):  
        yield (i*i)
```



Nest iteration

```
def all_pairs(x):
    for item1 in x:
        for item2 in x:
            yield(item1, item2)
```



Iterables

Demo



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



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



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
```



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...