Ch. 1 - The Python Data Model

How Special Methods Are Used

- meant to be called by Python interpreter and not programmer
- can be used to emulate numeric types with operator overloading
- can be used to set how a custom type is printed (**repr**)
    - **str** is called by the str() constructor and implicitly used by the print function
    - **str** should return a string suitable for display to end users
    - choose **repr** if only implementing one because when no custom **str** is available Python will call **repr** as a fallback
- can be used for arithmetic operators (**add** , **mul**)
- can be used to use custom type as a bool value (**bool**)

Overview of Special Methods

- string/bytes representation
    - `__repr__`
    - `__str__`
    - `__format__`
    - `__bytes__`
- conversion to number
    - `__abs__`
    - `__bool__`
    - `__complex__`
    - `__int__`
    - `__float__`
    - `__hash__`
    - `__index__`
- collection-like functionality
    - `__len__`
    - `__getitem__`
    - `__setitem__`
    - `__delitem__`
    - `__contains__`
- iteration
    - `__iter__`

    - `__reversed__`
    - `__next__`
- making an object callable
    - `__call__`
- context management
    - `__enter__`
    - `__exit__`
- instance creation and destruction
    - `__new__`
    - `__init__`
    - `__del__`
- attribute management
    - `__getattr__`
    - `__getattribute__`
    - `__setattr__`
    - `__delattr__`
    - `__dir__`
- attribute descriptors
    - `__get__`
    - `__set__`
    - `__delete__`
- class services
    - `__prepare__`
    - `__instancecheck__`
    - `__subclasscheck__`

Operators

- unary numeric operators

    - `__neg__`
        - `-`
    - `__pos__`
        - `+`
    - `__abs__`
        - `abs()`

- rich comparison operators

    - `__lt__`
        - `<`

    - `__le__`
        - `<=`
    - `__eq__`
        - `==`
    - `__ne__`
        - `!=`
    - `__gt__`
        - `>`
    - `__ge__`
        - `>=`
- arithmetic operators
    - `__add__`
        - `+`
    - `__sub__`
        - `-`
    - `__mul__`
        - `*`
    - `__truediv__`
        - `/`
        - from 2.2 - 2.*, 5 / 2 and 5 // 2 (integer) are equivalent unless adding `from __future__ import division` (with integers)
        - for 3 and with the future import above, returns 2.5
    - `__floordiv__`
        - always returns 2.0 (float) for 5 // 2
        - `//`
    - `__divmod__`
        - `%`
    - `__pow__`
        - `** or pow()`
    - `__round__`
        - `round()`
- reversed arithmetic operators
    - `__radd__`
    - `__rsub__`
    - `__rmul__`
    - `__rtruediv__`
    - `__rfloordiv__`
    - `__rmod__`

  - `__rdivmod__`
  - `__rpow__`
- augmented
  - `__iadd__`
  - `__isub__`
  - `__imul__`
  - `__itruediv__`
  - `__ifloordiv__`
  - `__imod__`
  - `__ipow__`
- bitwise operators
  - `__invert__`
    - `~`
  - `__lshift__`
    - `<<`
  - `__rshift__`
    - `>>`
  - `__and__`
    - `&`
  - `__or__`
    - `|`
  - `__xor__`
    - `^`
- reversed bitwise operators
  - `__rlshift__`
  - `__rrshift__`
  - `__rand__`
  - `__ror__`
  - `__rxor__`
- augmented bitwise operators
  - `__ilshift__`
  - `__irshift__`
  - `__iand__`
  - `__ior__`
  - `__ixor__`

- len is not called as a method because it gets special treatment as part of the Python data model, just like abs
- can also make len work with custom objects using the special method **len**
- special methods allow custom objects to behave like the built-in types
  - enables the expressive coding style the community considers Pythonic
- A basic requirement for a Python object is to provide usable string representations of itself, one used for debugging and logging, another for presentation to end users
  - purpose of special methods **repr** and **str** exist in the data model
- emulating sequences is one of the most widely used applications of the special methods
- Python offers a rich selection of numeric types, aided by operator overloading, from the built-ins to decimal.Decimal and fractions.Fraction, all supporting infix arithmetic operators
- Python data model - used in documentation
  - Python object model - used by many authors
- the Ruby community calls their equivalent of the special methods magic methods.
- the term metaobject protocol is useful to think about the Python data model and similar features in other languages
- metaobject refers to the objects that are the building blocks of the language itself
- in this context, protocol is a synonym of interface
- metaobject protocol ~= object model - an API for core language constructs.
- aspect-oriented programming is much easier to implement in a dynamic language like Python, and several frameworks do it, but the most important is zope.interface

II. Data Structures

2. An Array of Sequences

- mastering the standard library sequence types is a prerequisite for writing concise, effective, and idiomatic Python code

Overview of Built-In Sequences

- mutable
    - list, bytearray, array.array, collections.deque, and memoryview
- immutable
    - tuple, str, and bytes
- flat sequences
    - str, bytes, bytearray, memoryview, and array.array hold items of one type
    - physically store the value of each item within its own memory space, and not as distinct objects
    - more compact, faster, and easier to use
    - limited to storing atomic data such as numbers, characters, and bytes
- container sequences
    - list, tuple, and collections.deque can hold items of different types
    - hold references to the objects they contain, which may be of any type
    - more flexible
    - act surprisingly when they hold mutable objects
    - use with caution with nested data structures

```
Container       Iterable        Sized
__contains__    __iter__        __len__

^               ^               ^
Sequence
__getitem__
__contains__
__iter__
__reversed__
index
count

^
MutableSequence
__setitem__
__delitem__
insert
append
reverse
extend
pop
remove
__iadd__
```

List Comprehensions and Generator Expressions

- powerful notations to build and initialize sequences, master them
- listcomps and genexps

```
for symbol in symbols:
    codes.append(ord(symbol))

codes = [ord(symbol) for symbol in symbols]
```

- syntax tip - in Python code, line breaks are ignored inside pairs of [], {}, or ()
    - can build multiline lists, listcomps, genexps, dictionaries and the like without using the ugly \ line continuation escape
- in Python 2.x, variables assigned in the for clauses in list comprehensions were set in the surrounding scope
    - the value of outer variables can be overwritten
- prefer listcomps over builtin map and filter
    - may be faster in some cases but not always
    - test
- to initialize tuples, arrays, and other types of sequences can use listcomp
    - genexp saves memory because it yields items one by one using the iterator protocol instead of building a whole list just to feed another constructor
- use the same syntax as listcomps, but are enclosed in parentheses rather than brackets

```
tuple( ord( symbol) for symbol in symbols)
```

Tuples Are Not Just Immutable Lists

- tuples in Python play two roles
    - records with unnamed fields
    - immutable lists
- when a tuple is used as a record, tuple unpacking is the safest, most readable way of getting at the fields
- the new * syntax makes tuple unpacking even better by making it easier to ignore some fields and to deal with optional fields
- can use tuple unpacking with any iterable
    - tuple unpacking -> iterable unpacking

```
a, b, *rest = range( 5)
(0, 1, [2, 3, 4])
```

- can use nested tuple unpacking

```
for name, cc, pop, (latitude, longitude) in metro_areas:
```

- named tuples
    - like tuples, they have very little overhead per instance
    - provide convenient access to the fields by name and a handy ._asdict() to export the record as an OrderedDict.
- tuples as immutable lists
    - tuple supports all list methods that do not support adding or removing items
        - except **reverse** but reversed(my_tuple) works without it
    - **add**
    - **iadd**
    - append
    - clear
    - **contains**
    - copy
    - count
    - **delitem**
    - extend
    - **getitem**
    - **getnewargs**
    - index
    - insert
    - **iter**
    - **len**
    - **mul**
    - **imul**
    - **rmul**
    - pop
    - remove
    - reverse
    - **reversed**
    - **setitem**

    - sort

Slicing

- sequence slicing is a favorite Python syntax feature, and it is even more powerful than many realize
- multidimensional slicing and ellipsis (...) notation, as used in NumPy, may also be supported by user-defined sequences
- assigning to slices is a very expressive way of editing mutable sequences
- repeated concatenation as in seq * n is convenient and, with care, can be used to initialize lists of lists containing immutable items

Using + and * with Sequences

Augmented Assignment with Sequences

- augmented assignment with + = and *= behaves differently for mutable and immutable sequences
    - *= - these operators necessarily build new sequences
    - if the target sequence is mutable, it is usually changed in place — but not always, depending on how the sequence is implemented

list.sort and the sorted Built-In Functions

- the sort method and the sorted built-in function are easy to use and flexible, thanks to the key optional argument they accept, with a function to calculate the ordering criterion
- key can also be used with the min and max built-in functions

Managing Ordered Sequences with bisect

- to keep a sorted sequence in order, always insert items into it using bisect.insort; to search it efficiently, use bisect.bisect

When a List Is Not the Answer

- Python standard library provides array.array.
- NumPy and SciPy are not part of the standard library but should study
- thread-safe collections.deque
    - comparing its API with that of list in Table 2-3 and mentioning other queue implementations in the standard library

3. Dictionaries and Sets

- Python dicts are highly optimized
- hash tables are the engines behind Python's high-performance dicts

Generic Mapping Types

- collections.abc
    - Mapping
    - MutableMapping
    - formalize the interfaces of dict
- often extend collections.UserDict instead

```
Container       Iterable        Sized
__contains__    __iter__        __len__

^               ^               ^
Mapping
__getitem__
__contains__
__eq__
__ne__
get
items
values

^
MutableMapping
__setitem__
__delitem__
clear
pop
popitem
setdefault
update
```

- keys must be hashable
    - an object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an **eq** () method)
    - hashable objects which compare equal must have the same hash value
    - the atomic immutable types (str, bytes, numeric types) are all hashable
    - a frozenset is always hashable, because its elements must be hashable by definition
    - a tuple is hashable only if all its items are hashable

- user-defined types are hashable by default because their hash value is their id () and they all compare not equal
    - if an object implements a custom `__eq__` that takes into account its internal state it may be hashable only if all its attributes are immutable

dict Comprehensions

- dictcomp added in Python 2.7

```
country_code = {country: code for code, country in DIAL_CODES}
```

Overview of Common Mapping Methods

- defaultdict and OrderedDict in collections
    - clear
    - **contains**
    - copy
    - **copy**
    - default_factory
    - **delitem**
    - fromkeys
    - get
    - **getitem**
    - items
    - **iter**
    - keys
    - **len**
    - **missing**
    - move_to_end
    - pop
    - popitem
    - **reversed**
    - setdefault
    - **setitem**
    - update
    - values

Handling Missing Keys with setdefault

- dict access with `d[k]` raises an error when k is not an existing key

- d.get(k, default) is an alternative to `d[k]` whenever a default value is more convenient than handling KeyError
- when updating the value found (if it is mutable), using either **getitem** or get is awkward and inefficient

```
my_dict.setdefault(key, []).append(new_value)

# same as
if key not in my_dict:
    my_dict[key] = []
my_dict[ key]. append( new_value)
```

- setdefault uses a single lookup whereas the second example uses at least two and three if its not found

Mappings with Flexible Key Lookup

Ch. 2 - An Array of Sequences

Overview of Built-In Sequences

- container sequences - items of different types (references)
    - list
    - tuple
    - collections.deque
- flat sequences - items of one type (physically store value in memory space)
    - str
    - bytes
    - bytearray
    - memoryview
    - array.array
- mutable
    - list
    - bytearray
    - array.array
    - collections.deque
    - memoryview

- immutable
    - tuple
    - str
    - bytes
- inheritance (see diagram for methods)

```
MutableSequences -> Sequence -> Container
                             -> Iterable
                             -> Sized
```

List Comprehensions and Generator Expressions

- listcomps and genexps often faster and more readable
- tip - can build multiline lists, listcomps, and genexps within `[], {}, or ()`
- listcomps previously leaked variable names into the surrounding scope (e.g. overrode variable values of the same variable name)
    - fixed in Python 3
- map and filter are not necessarily faster

```
# arranged by color then size
[(color, size) for color in colors for size in sizes]

# arranged in same order as list from above
for color in colors:
    for size in sizes:
        print((color, size))

# ordered by size as the second element then color
[(color, size) for size in sizes for color in colors]
```

- listcomps only build lists
    - use genexps to fill up sequence types
- genexp
    - can build tuple, array, and other types of sequences
    - enclosed in parens rather than brackets
    - saves memory
        - yields items one by one using the iterator protocol instead of building a whole list just to feed another constructor

```
# no paren duplication
tuple(ord( symbol) for symbol in symbols)

# paren duplication due to multi-arg ctor
array.array('I', (ord( symbol) for symbol in symbols))
```

- can use in for loops

```
for tshirt in ('% s %s' % (c, s) for c in colors for s in sizes):
    print( tshirt)
```

- saves the expense of building lists to feed the for loop

Tuples Are Not Just Immutable Lists

- immutable lists and records with no field names
- as records
    - each item in the tuple holds the data for one field and the position of the item gives its meaning
- tuple unpacking can be done
    - to assign elements to variables
    - to assign elements to a format string
    - works with any iterable object
        - only requirement is iterable yields one item per variable in the receiving tuple
        - unless using * to capture excess items
    - tuple unpacking == iterable unpacking
    - can use to unpack as args to function
- can use to swap

```
b, a = a, b
```

- can use as return value from function
- can use _ as a placeholder variable but be careful when writing internationalized software (used as an alias to gettext.gettext)
- can use * for excess items, and can appear in any position

```
a, b, *rest = range(5)
```

- works with nested structures
    - could define functions with nested tuples in formal parameters in Python 2
    - cannot in Python 3

- use collections.namedtuple for debugging and more
    - same memory consumed as regular tuple and less than a class
    - two parameters to create a named tuple
        - class name
        - list of field names
            - iterable of strings
            - single space-delimited string
- tuples as immutable lists
    - see table for list of similar functionality

Slicing

- slices and range use [0,n) range functionality
    - easy to see length of a slice or range when only the stop position is given
        - `range(3)` and `list[:3]` produce three items
    - easy to compute the length of a slice or range when start and stop are given
        - subtract stop - start
    - easy to split a sequence in two parts at any index x without overlap
        - `list[:x] and list[x:]`
- stride for third element causes skip or items in reverse
    - uses `slice(start, stop, step)` object and a seq calls `seq.__getitem__(slice(start, stop, step))`
- can use slice objects (notation for `seq[start:stop:step]` is only valid in `[]`)

```
s1 = slice(0, 6)
elems = items[s1]
```

- `[]` operator can take multiple indices or slices separated by comma

    - can use with numpy.ndarray to get 2-dim slices
    - calls `a.__getitem__(( i, j))` to evaluate `a[i, j]`

- `...` is a token and is an Ellipsis object - can be used to represent `[:]` in intermediate dimensions of a multi-dimensional array/list

- slices can be used to assign/modify and delete (using `del`)

```
l = list(range(10))
l[2:5] = [20, 30]
del l[5:7]
l[3::2] = [11, 22]
l[2: 5] = 100  # error - needs iterable
l[2:5] = [100]  # just assigns 100 to index 2
```

Using + and * with Sequences

- operands to + and * for sequences must be of the same sequence type and produce a new sequence without modifying the originals
- expressions like a * n when a is a sequence containing mutable items have unexpected results
  - trying to initialize a list of lists as `l = [[]] * 3` results in a list with three references to the same inner list
- list of lists

```
# proper method
[['_'] * 3 for i in range(3)]

# equivalent to
board = []
for i in range(3):
    row = ['_'] * 3
    board.append(row)



# list with references to the same lists
[['_'] * 3] * 3

# equivalent to
row = ['_'] * 3
board = []
for i in range(3):
    board.append(row)
```

Augmented Assignment with Sequences

- += and *= operators behave differently depending on the mutability of the first argument
- special method for += is `__iadd__` (in-place add) with a fallback to `__add__` when it isn't implemented
- for mutable sequences `__iadd__` is most likely implemented and += most likely happens in place
  - in place is not possible for immutable sequences
- *= -> `__imul__`
  - for mutable - after multiplication the sequence is the same object with items appended
  - for immutable - after multiplication a new sequence is created

- repeated concatenation of immutable sequences is inefficient because of a large number of copies in the interpreter

- NOTE: can use dis module to disassemble Python code (for CPython)!!!

- corner case example lessons

  - do not put mutable items in tuples
  - augmented assignment is not an atomic operation
    - can throw an exception after doing part of the operation
  - can easily inspect Python bytecode
    - often helpful to see what is going on under the hood

list.sort and the sorted Built-In Function

- list.sort sorts in place and returns None
  - convention - functions or methods that modify in-place should return None
  - cannot cascade calls to these methods
    - see Wiki Fluent Interface
- sorted creates a new list and returns it
  - accepts any iterable object and always returns a sorted list
- both list.sort and sorted take two optional keyword args
  - reverse - True or False for reversed order sort
  - key - one arg function used to produce the sorting key (default is compare items)
    - can also be used with the min() and max() built-ins and more

Managing Ordered Sequences with bisect

- standard binary search in `bisect` module
  - haystack - first arg and sorted sequence (ascending order)
  - needle - second arg to search for
- can use the result of bisect(haystack, needle) as the index argument to haystack.insert(index, needle)
  - using insort does both steps, and is faster
- see https://code.activestate.com/recipes/577197-sortedcollection/ for an easy to use sorted collection
- can fine-tune bisect behavior
  - pair of optional arguments lo and hi can be used to narrow the region in the sequence to be searched when inserting
    - lo defaults to 0
    - hi to len() of the sequence

    - bisect is an alias for bisect_right
        - sister function called bisect_left
        - difference is apparent only when the needle compares equal to an item in the list
            - bisect_right returns an insertion point after the existing item
            - bisect_left returns the position of the existing item so insertion would occur before it
            - if the sequence contains objects that are distinct yet compare equal, then it may be relevant
- bisect.insort(seq, item) can be used to insert new items in a sorted sequence in sorted order
    - takes optional lo, hi
    - also has insort_left

When a List is Not the Answer

- better options than list for specific use cases
- array is better for storing large amounts of numeric data
    - does not hold full numeric objects
    - holds packed byte representation of values
- collections.deque is better for constantly adding and removing data from the ends of a sequence
- array supports all mutable sequence types and additional methods for fast loading (`.frombytes` and `.tofile`)
    - as minimal as a C-style array
- `.tofile` and `.fromfile` are very fast (nearly 60x and 7x) and saves a large amount of disk memory
- can use pickle also (not as fast for numeric data, but handles almost all built-in types)
- use bytes and bytearray for numeric arrays that represent binary data (images, etc)
- see Table 2-2 for comparison of the features of list and array.array
- array does not have an `array.sort` - use the sorted built-in and bisect.insort

```
a = array.array( a.typecode, sorted( a))
```

- built-in memoryview class
    - shared-memory sequence type that allows handling slices of arrays without copying bytes
    - essentially a generalized NumPy array structure in Python itself (without the math)
    - allows sharing memory between data-structures (things like PIL images, SQLlite databases, NumPy arrays, etc.) without copying
    - very important for large data sets

    - memoryview.cast method allows changing the way multiple bytes are read or written as units without moving bits around (just like the C cast operator)
        - returns another memoryview object, always sharing the same memory

```
numbers = array.array('h', [-2, -1, 0, 1, 2])
memv = memoryview(numbers)
len(memv) 5
memv[0] -2
memv_oct = memv.cast('B')
```

- use numpy and scipy for complex numerical/scientific operations on sequences types
    - `numpy.ndarray`
- can use a list as a stack or queue using `.append` or `.pop` but inserting or removing from the left (0 index) of a list is costly
    - use collections.deque
- deque is thread-safe and double-ended
    - can also be bounded
    - implements most of the list methods, and adds a few specific to its design, like popleft and rotate
    - hidden cost - removing items from the middle of a deque is not as fast
        - optimized for appending and popping from the ends
    - append and popleft are atomic so safe for multithreaded applications
- see table 2-3 for features of deque
- additional modules for list-like data structures
    - queue
        - provides the synchronized (i.e., thread-safe) classes Queue, LifoQueue, and PriorityQueue
        - used for safe communication between threads
        - can be bounded by providing a maxsize argument greater than 0 to the constructor
        - don't discard items to make room as deque does
            - when the queue is full the insertion of a new item blocks — i.e., it waits until some other thread makes room by taking an item from the queue
            - useful to throttle the number of live threads
    - multiprocessing
        - bounded Queue (similar to queue.Queue) but designed for interprocess communication
        - specialized multiprocessing.JoinableQueue is also available for easier task management

- asyncio
    - New to Python 3.4
    - provides Queue, LifoQueue, PriorityQueue, and JoinableQueue with APIs inspired by the classes contained in the queue and multiprocessing modules
    - adapted for managing tasks in asynchronous programming
- heapq
    - heapq does not implement a queue class
    - provides functions like heappush and heappop
    - allow use of a mutable sequence as a heap queue or priority queue

Ch. 3 - Dictionaries and Sets

- much of the Python engine is implemented using dictionaries under the hood
- hash tables are used to implement Python dicts

Generic Mapping Types

- `collections.abc` provides Mapping and MutableMapping

```
MutableMapping -> Mapping -> Container
                          -> Iterable
                          -> Sized
```

- custom mapping types often extend dict or `collections.UserDict`
- best to check with

```
isinstance(instance, abc.Mapping)
```

- all std lib mapping types use dict and so keys must be hashable
- an object is hashable if it
    - has a hash value which never changes during its lifetime
        - needs a **hash** () method
    - can be compared to other objects
        - needs an **eq** () method
    - hashable objects which compare equal must have the same hash value
- atomic immutable types (str, bytes, numeric types) are all hashable
- frozenset is always hashable, because its elements must be hashable by definition
- tuple is hashable only if all its items are hashable
    - other than that, all immutable built-in objects are hashable

- user-defined types are hashable by default because their hash value is their id() and they all compare not equal
  - if an object implements a custom **eq** that takes into account its internal state, it may be hashable only if all its attributes are immutable

dict Comprehensions

- dictcomp builds a dict instance by producing key:value pair from any iterable

```
d = {key: val for key_v, value_v in seq}
```

Overview of Common Mapping Methods

- dict, defaultdict and OrderedDict (in collections)
- see table 3-1
- setdefault
  - dict access with `d[k]` raises an error when k is not an existing key
  - `d.get(k, default)` is an alternative to `d[k]` when a default value is more convenient than handling KeyError
  - when updating the value found (if it is mutable), using either `__getitem__` or `get` is awkward and inefficient
- setdefault only performs one lookup whereas `if key not in d:` performs 2 or 3 lookups

```
# prefer
d.setdefault(key, val)

# over
if key not in d:
```

Mappings with Flexible Key Lookup

- defaultdict is configured to create items on demand whenever a missing key is searched
  - on creation, user provides a callable that is used to produce a default value when `__getitem__` is passed a nonexistent key argument
  - e.g. `dd = defaultdict(list)` if 'new_key' is not in dd, `dd['new_key']`
    - calls list() to create a new list
    - inserts the list into dd using 'new_key' as key
    - returns a reference to that list

    - callable that produces the default values is held in an instance attribute called default_factory
    - uses `__missing__` to call default_factory
- `__missing__` special method
    - not defined in the base dict class
    - subclass dict and provide a `__missing__` method, the standard `dict.__getitem__` will call it whenever a key is not found, instead of raising KeyError
    - just called by `__getitem__` (i.e., for the `d[k]` operator)
        - presence of a `__missing__` method has no effect on the behavior of other methods that look up keys, such as get or `__contains__` (which implements the in operator)
        - why default_factory of defaultdict works only with `__getitem__`
- `k in my_dict.keys()` is efficient in Python 3 even for very large mappings because dict.keys() returns a view (similar to a set)
    - containment checks in sets are as fast as in dictionaries
        - see "Dictionary" view objects section of the documentation
    - in Python 2, dict.keys() returns a list which is not efficient for large dictionaries, because k in my_list must scan the list

Variations of dict

- collections.OrderedDict
    - maintains keys in insertion order, allowing iteration over items in a predictable order
    - popitem method of an OrderedDict pops the last item by default, but if called as my_odict.popitem(last = False), it pops the first item added
- collections.ChainMap
    - holds a list of mappings that can be searched as one
    - lookup is performed on each mapping in order, and succeeds if the key is found in any of them
    - useful to interpreters for languages with nested scopes, where each mapping represents a scope context
- collections.Counter
    - mapping that holds an integer count for each key
    - updating an existing key adds to its count
    - can be used to count instances of hashable objects (the keys) or as a multiset — a set that can hold several occurrences of each element
    - implements the + and - operators to combine tallies
    - has other useful methods such as most_common([n])
        - returns an ordered list of tuples with the n most common items and their counts
- collections.UserDict
    - pure Python implementation of a mapping that works like a standard dict

Subclassing UserDict

- prefer over dict
    - built-in dict has methods that are shortcuts
    - corresponding methods in UserDict work without begin overridden
- UserDict does not inherit from dict but uses dict for internal data attribute
    - undesired recursion when coding special methods like `__setitem__`
    - simplifies coding of `__contains__`
- methods to note
    - `__missing__`
    - `__contains__`
    - `__setitem__`
    - `MutableMapping.update`
    - `Mapping.get`
- see PEP 455 and TransformDict as well

Immutable Mappings

- Python 3.3 - `types.MappingProxyType` returns `mappingproxy` which is a dynamic read-only view into a mapping type

Set Theory

- set and frozenset - unique collections of objects
    - elements must be hashable
    - set is not hashable
    - frozenset is hashable so a set may contain frozensets
    - implement the set operations as infix operators
        - given two sets a and b
        - a | b returns their union
        - a & b computes the intersection
        - a - b the difference
    - use of set operations can reduce both the line count and the runtime of Python programs
- can write set literals as `{1, 2, 3}` but must use `set()` for an empty set
    - `{}` is an empty dict
    - set literal construction is faster and more readable than calling the constructor
- frozenset must be constructed with the constructor

```
frozenset([1, 2, 3, 4])
```

- setcomps can be constructed as dictcomps, using `{}`
- see table 3-2 for set operations and methods

dict and set Under the Hood

- dicts and sets are fast
  - searches are fastest in set, then dict, then list
- C code has many optimizations
- hash table
  - sparse array (i.e., an array that always has empty cells)
    - cells in a hash table are often called "buckets"
  - dict hash table
    - bucket for each item, and it contains two fields
      - a reference to the key
      - a reference to the value of the item
    - all buckets have the same size
    - access to an individual bucket is done by offset
  - Python tries to keep at least 1/ 3 of the buckets empty
    - if too dense, copied to resize
  - for insert, the first step is to calculate the hash value of the item key
    - done with the hash() built-in function
- hash() built-in function works directly with built-in types and falls back to calling `__hash__` for user-defined types
  - if two objects compare equal, their hash values must also be equal
  - otherwise the hash table algorithm does not work
  - starting with Python 3.3, a random salt value is added to the hashes of str, bytes, and datetime objects
    - salt value is constant within a Python process but varies between interpreter runs
    - random salt is a security measure to prevent a DOS attack
- for get
  - Python calls `hash(search_key)` to obtain the hash value of search_key and uses the least significant bits of that number as an offset to look up a bucket in the hash table
  - the number of bits used depends on the current size of the table
  - if the found bucket is empty, KeyError is raised
  - otherwise, the found bucket has an item
    - then Python checks whether search_key == found_key
    - on match the found value is returned

        - if search_key and found_key do not match, this is a hash collision
            - happens because a hash function maps arbitrary objects to a small number of bits and the hash table is indexed with a subset of those bits
        - to resolve the collision
            - algorithm then takes different bits in the hash
            - massages them in a particular way, and uses the result as an offset to look up a different bucket
            - if empty KeyError is raised
            - otherwise return item value or repeat collision resolution
    - same process for insert or update
- consequences
    - keys must be hashable objects
        - supports the hash() function via a `__hash__()` method that always returns the same value over the lifetime of the object
        - supports equality via an `__eq__()` method
        - if a == b is True then hash(a) == hash(b) must also be True
    - dicts have significant memory overhead
        - must be sparse
        - better to use list of tuples or named tuples
            - removes overhead of one hash table per record
            - does not store each field name again with each record
    - key search is very fast
        - trades space for time
    - key ordering depends on insertion order
        - when a hash collision happens, the second key ends up in a position that it would not normally occupy if it had been inserted first
        - dict built as dict([(key1, value1), (key2, value2)]) compares equal to dict([(key2, value2), (key1, value1)])
        - key ordering may not be the same if the hashes of key1 and key2 collide
    - adding items to a dict may change the order of existing keys
- practical consequences of sets
    - also use hash tables
    - dict consequences above apply
        - set elements must be hashable objects
        - sets have a significant memory overhead
        - membership testing is very efficient
        - element ordering depends on insertion order
        - adding elements to a set may change the order of other elements

Ch. 4 - Text versus Bytes

- definition of character complicates definition of string
    - character is generally Unicode character
    - Python 3
        - `bytes` is raw bytes
        - `str` is Unicode chars
    - Python 2
        - `str` is raw bytes
        - `unicode` obj is Unicode chars
- Unicode standard
    - identity of a character (code point)
        - number from 0 to 1,114,111 (base 10)
        - shown in the Unicode standard as 4 to 6 hexadecimal digits with a "U +" prefix
        - about 10% of the valid code points have characters assigned to them in Unicode 6.3, the standard used in Python 3.4
    - encoding is an algorithm that converts code points to byte sequences and vice versa
        - actual bytes that represent a character depend on the encoding in use
        - code point for A (U + 0041) is encoded as the single byte \x41 in the UTF-8 encoding,
    - converting from code points to bytes is encoding
    - converting from bytes to code points is decoding
- Python 3 str is pretty much the Python 2 unicode type
- Python 3 bytes is not the old str renamed
    - closely related bytearray type

Byte Essentials

- there are two basic built-in types for binary sequences

    - immutable bytes type introduced in Python 3
    - mutable bytearray, added in Python 2.6
    - Python 2.6 also introduced bytes
        - just an alias to the str type
        - does not behave like the Python 3 bytes type

- each item in bytes or bytearray is an integer from 0 to 255

    - not a one-character string like in the Python 2 str

- a slice of a binary sequence always produces a binary sequence of the same type

- `b[0]` returns an int but `b[:1]` returns a bytes object of length 1
    - only sequence type where `s[0] == s[:1]` is the str type
    - for every other sequence, `s[i]` returns one item, and `s[i:i + 1]` returns a sequence of the same type with the `s[i]` item inside it
- are really sequences of integers, their
- literal notation of binary sequences (as integers) reflects the fact that ASCII text is often embedded in them
    - three different displays are used, depending on each byte value
        - bytes in the printable ASCII range the ASCII character itself is used
        - bytes corresponding to tab, newline, carriage return, and , the escape sequences \t, \n, \r, and \ are used
        - for every other byte value, a hexadecimal escape sequence is used (e.g., \x00 is the null byte)
- both bytes and bytearray support every str method
    - except formatting (format, format_map)
    - others that depend on Unicode data, including casefold, isdecimal, isidentifier, isnumeric, isprintable, and encode
- regular expression functions in the re module also work on binary sequences, if the regex is compiled from a binary sequence instead of a str
- % operator does not work with binary sequences in Python 3.0 to 3.4 but is beyond
- has `.fromhex` method
- means of construction
    - str and an encoding keyword argument
    - iterable providing items with values from 0 to 255
    - single integer, to create a binary sequence of that size initialized with null bytes (deprecated in Python 3.5 and removed in Python 3.6)
    - an object that implements the buffer protocol (e.g., bytes, bytearray, memoryview, array.array)
        - copies the bytes from the source object to the newly created binary sequence
- struct module provides functions to parse packed bytes into a tuple of fields of different types and to perform the opposite conversion, from a tuple into packed bytes
    - used with bytes, bytearray, and memoryview objects

- memoryview class does not allow creation or storage of byte sequences
  - provides shared memory access to slices of data from other binary sequences, packed arrays, and buffers such as Python Imaging Library (PIL) images, without copying the bytes
  - slicing returns a new memoryview without copying bytes
- NOTE: use mmap for memory-mapped files

Basic Encoders/Decoders

- bundles more than 100 codecs with aliases
- latin1 a.k.a. iso8859_1
  - important because it is the basis for other encodings, such as cp1252 and Unicode itself (note how the latin1 byte values appear in the cp1252 bytes and even in the code points)
- cp1252
  - a latin1 superset by Microsoft
  - adds useful symbols like curly quotes and the € (euro)
  - some Windows apps call it "ANSI"
  - never a real ANSI standard
- cp437
  - original character set of the IBM PC, with box drawing characters
  - incompatible with latin1, which appeared later
- gb2312
  - legacy standard to encode the simplified Chinese ideographs used in mainland China
  - one of several widely deployed multibyte encodings for Asian languages
- utf-8
  - most common 8-bit encoding on the Web
  - backward-compatible with ASCII (pure ASCII text is valid UTF-8)
- utf-16le
  - one form of the UTF-16 16-bit encoding scheme
  - all UTF-16 encodings support code points beyond U + FFFF through escape sequences called "surrogate pairs"

Understanding Encode/Decode Problems

- generic UnicodeError exception
- reported error is almost always more specific
  - UnicodeEncodeError - when converting str to binary sequences
  - UnicodeDecodeError - when reading binary sequences into str

- can get a SyntaxError when loading Python modules when the source encoding is unexpected . We'll show how to handle all of these errors in the next sections. Tip The first thing to note
- when receiving a Unicode error, note the exact type of the exception
- dealing with UnicodeEncodeError
  - most non-UTF codecs handle only a small subset of the Unicode characters
  - can use `error='ignore'` (generally bad to do) in encode call
  - can use `error='replace'` or `errors =' xmlcharrefreplace'` to automatically replace in encode call
- dealing with UnicodeDecodeError
  - not every byte holds a valid ASCII character
  - not every byte sequence is valid UTF-8 or UTF-16
  - many legacy 8-bit encodings like 'cp1252', 'iso8859_1', and 'koi8_r' are able to decode any stream of bytes, including random noise, without generating errors
  - if the program assumes the wrong 8-bit encoding, it will silently decode garbage
  - can also pass `errors='replace'` here
- dealing with SyntaxError when loading modules with unexpected encoding
  - UTF-8 is the default source encoding for Python 3
  - ASCII was the default for Python 2 (starting with 2.5)
  - loading a .py module containing non-UTF-8 data and no encoding declaration
  - UTF-8 is widely deployed in GNU/Linux and OSX systems
    - possible to open a .py file created on Windows with cp1252
  - error happens in Python for Windows, because the default encoding for Python 3 is UTF-8 across all platforms
  - add a magic coding comment at the top of the file to fix this problem `# coding: cp1252`
  - Python 3 source code is no longer limited to ASCII and defaults to the excellent UTF-8 encoding, the best "fix" for source code in legacy encodings like 'cp1252' is to convert them to UTF-8 already, and not bother with the coding comments. If your editor does not support UTF-8, it's time to switch.
- must be told what the encoding of a byte sequence is
  - can use libraries like Chardet to determine within a confidence level
- BOM - byte order mark or bytes denoting endianness of CPU where encoding was performed

Handling Text Files

- best practice for handling text

  - bytes should be decoded to str as early as possible on input (e.g., when opening a file for reading)

- text handling should be done exclusively on str objects within business logic of program
    - never encode or decode in the middle of other processing
- str objects should be encoded to bytes as late as possible on output

- in Python 3, open built-in handles the decoding and encoding on read and write

- code that has to run on multiple machines should never depend on encoding defaults

    - always pass an explicit encoding argument when opening text files
    - default may change from one machine to the next

- do not open text files in binary mode unless analyzing the file contents to determine encoding

    - prefer Chardet for analyzing the the file contents

- encoding defaults

```
locale.getpreferredencoding()
type(my_file)
my_file.encoding
sys.stdout.isatty()
sys.stdout.encoding
sys.stdin.isatty()
sys.stdin.encoding
sys.stderr.isatty()
sys.stderr.encoding
sys.getdefaultencoding()
sys.getfilesystemencoding()
```

- Linux and macOS returned same results, Windows was very different
    - Linux and macOS are set to UTF-8 by default
    - locale.getpreferredencoding() is the most important setting
    - text files use locale.getpreferredencoding() by default
    - output is going to the console, so sys.stdout.isatty() is True
        - sys.stdout.encoding is the same as the console encoding
- if the encoding argument is omitted when opening a file, default is given by locale.getpreferredencoding()
- encoding of sys.stdout/stdin/stderr is given by PYTHONIOENCODING env var
    - if not set, inherited from console or defined by locale.getpreferredencoding() if IO is going to/from a file
- sys.getdefaultencoding() is used internally to convert binary to/from str
    - setting cannot be changed
- sys.getfilesystemencoding() is used to encode/decode filenames (but not file contents)
- NOTE: do not rely on encoding defaults!!!

Normalizing Unicode for Saner Comparisons

- Unicode has combining chars like diacritics
- use unicodedata.normalize
    - on of four strings 'NFC', 'NFD', 'NFKC', 'NFKD' as first arg
    - normalization Form C (NFC) composes the code points to produce the shortest equivalent string
    - NFD decomposes, expanding composed characters into base characters and separate combining characters
    - NFKC and NFKD — the letter K stands for "compatibility"
        - stronger forms of normalization, affecting "compatibility characters"
        - some Unicode characters appear more than once for compatibility with preexisting standards
        - each compatibility character is replaced by a "compatibility decomposition" of one or more characters that are considered a preferred representation
        - done even if there is some formatting loss
    - use NFKC and NFKD only for special cases like search and indexing because they cause data loss
- can use `str.casefold()` which returns different characters for 116 of 110,122 named characters in Unicode 6.3

```
from unicodedata import normalize
def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() == normalize('NFC', str2).casefold()
```

- Google search ignores diacritics in certain contexts

```
def remove_diacritics(txt):
    norm_txt = unicodedata.normalize('NFD', txt)
    shaved = ''.join(c for c in norm_txt if not unicodedata.combining(c))
    return unicodedata.normalize('NFC', shaved)
```

- see text for more moderate examples
    - different languages have different rules for removing diacritics

Sorting Unicode Text

- Python sort compares items one-by-one, e.g. code points for strings
  - causes issues when using non-ASCII chars
- use local.strxfrm to make local-aware comparisons
  - must first set the local using `local.setlocal()`
  - calling setlocale in a library is not recommended because locale settings are global
    - application or framework should set the locale when the process starts , and should not change it afterwards.
  - locale must be installed on the OS, otherwise setlocale raises a locale.Error: unsupported
  - local name must be spelled correctly
    - They are pretty much
    - standardized in the Unix derivatives as 'language_code.encoding',
    - syntax is more complicated on Windows - Language Name-Language Variant_Region Name.codepage
  - local must be correctly implemented by OS (incorrect on macOS)
- can use PyUCA

The Unicode Database

- DB mapping code points to character names and metadata about relationships between characters
- `unicodedata` module

Dual-Mode str and bytes APIs

- re and os modules (plus more) have functions that accept str or bytes
- \d, \w, etc used in regexes with bytes only match ASCII
  - when used with str matches Unicode digits and letters beyond ASCII
- many filenames on Linux may have byte sequences that cannot be decoded to str
  - common on servers with clients from a variety of OSes
- all os module functions that accept filenames or pathnames take arguments as str or bytes
  - called with a str argument
    - argument will be automatically converted using the codec named by sys.getfilesystemencoding()
    - OS response will be decoded with the same codec
  - can pass bytes arguments to the os functions to get bytes return values

- fsencode(filename)
    - encodes filename (can be str or bytes) to bytes using the codec named by sys.getfilesystemencoding() if filename is of type str, otherwise returns the filename bytes unchanged
- fsdecode(filename)
    - decodes filename (can be str or bytes) to str using the codec named by sys.getfilesystemencoding() if filename is of type bytes, otherwise returns the filename str unchanged . On Unix-derived platforms,
- these functions use the surrogateescape error handler to avoid issues with unexpected bytes on Unix systems
- strict error handler is used on Windows

Ch. 5 - First-Class Functions

- first-class object is a program entity that can be
    - created at runtime
    - assigned to a variable or element in a data structure
    - passed as an argument to a function
    - returned as the result of a function

Treating a Function Like an Object

- can use `__doc__` attribute for help text and call `help(func)`
- functions also have other accessible attributes
- can assign functions to vars and call
- can pass as args to other functions

Higher-Order Functions

- function that takes other functions as arguments, e.g. map, sorted
- functional paradigm
    - map
    - filter
    - reduce
    - apply (deprecated in favor of *args and **kwargs)
- prefer listcomps, gencomps, etc over map and filter
- map & filter return generators in Python 3
- reduce was moved from built-in to functools
    - prefer sum

- all(iterable)
  - returns True if every element of the iterable is truthy
  - `all([])` returns True
- any(iterable)
  - returns True if any element of the iterable is truthy
  - `any([])` returns False

Anonymous Functions

- use lambda
  - body must be pure, e.g. cannot make assignments or use statements such as while, try, etc.
- rarely useful
- see https://docs.python.org/3/howto/functional.html

The Seven Flavors of Callable Objects

- use the `callable()` built-in to determine if an object is callable
- callable types
  - user-defined functions
    - created with def statements or lambda expressions
  - built-in functions
    - a function implemented in c (for CPython), like len or time.strftime
  - built-in methods
    - methods implemented in C, like dict.get. methods functions defined in the body of a class
  - classes
    - when invoked, a class runs its **new** method to create an instance, then **init** to initialize it, and finally the instance is returned to the caller
    - because there is no new operator in python, calling a class is like calling a function (usually calling a class creates an instance of the same class, but other behaviors are possible by overriding **new**)
  - class instances
    - if a class defines a **call** method, then its instances may be invoked as functions
  - generator functions
    - functions or methods that use the yield keyword
    - when called, generator functions return a generator object
    - generator functions are unlike other callables in many respects

User-Defined Callable Types

- instances of Python objects can be made to behave like functions by implementing a `__call__` instance method
- good for storing state across invocations
- also useful for decorators that need to store state (for example memoization)
- see also closures

Function Introspection

- use `dir` built-in to see attributes
- uses `__dict__` to store user attributes assigned to it
- not common to assign attributes to functions, but is a practice used by Django
- attributes specific to functions

Name	Type	Description
annotations	dict	parameter and return annotations
call	method-wrapper	implementation of the () operator; a.k.a. the callable object protocol
closure	tuple	the function closure, i.e., bindings for free variables (often is None)
code	code	function metadata and function body compiled into bytecode
defaults	tuple	default values for the formal parameters
get	method-wrapper	implementation of the read-only descriptor protocol
globals	dict	global variables of the module where the function is defined
kwdefaults	dict	default values for the keyword-only formal parameters
name	str	the function name
qualname	str	the qualified function name, e.g., Random.choice (see PEP-3155)

From Positional to Keyword-Only Parameters

- keyword only args and * and ** to "explode" args

- prefixing an argument with * captures any number of arguments as a tuple
- prefixing an argument with ** captures any number of arguments as a dict
- can support keyword only positional arguments with f(a, *, b)
    - keyword-only positional arguments can be required by not defining a default value for it

Retrieving Information About Parameters

- `__defaults__` attribute holds a tuple with the default values of positional and keyword arguments
- defaults for keyword-only arguments appear in `__kwdefaults__`
- names of the arguments are found within the `__code__` attribute
    - a reference to a code object with many attributes of its own
- `__defaults__` , `__code__.co_varnames` , and `__code__.co_argcount`
- see examples 5-16 and 5-17 for extracting function arg info
- inspect.signature returns an inspect.Signature object
    - has a parameters attribute
        - can read an ordered mapping of names to inspect.Parameter objects
    - each Parameter instance has attributes such as name, default, and kind
    - special value inspect._empty denotes parameters with no default
    - kind attribute holds one of five possible values from the _ParameterKind class
        - POSITIONAL_OR_KEYWORD
            - a parameter that may be passed as a positional or as a keyword argument (most Python function parameters are of this kind).
        - VAR_POSITIONAL
            - a tuple of positional parameters
        - VAR_KEYWORD
            - a dict of keyword parameters
        - KEYWORD_ONLY
            - a keyword-only parameter (new in Python 3).
        - POSITIONAL_ONLY
            - a positional-only parameter
            - currently unsupported by Python function declaration syntax, but shown by existing functions implemented in C — like divmod — that do not accept parameters passed by keyword
    - also has an annotation attribute that is usually inspect._empty but may contain function signature metadata provided via the new annotations syntax in Python 3

Function Annotations

- Python 3 special syntax to attach metadata to the parameters of a function declaration and its return value

- annotation goes between the argument name and the = sign with default values
- add -> and another expression between the) and the : at the tail of the function declaration to annotate the return value
- expressions may be of any type
- most common types used in annotations are classes, like str or int, or strings, like 'int > 0'
- no processing is done with annotations
  - merely stored in the **annotations** attribute of the function
- only thing Python does with annotations is to store them in the `__annotations__` attribute of the function
  - annotations have no meaning to the Python interpreter
  - metadata that may be used by tools, such as IDEs, frameworks, and decorators

Packages for Functional Programming

- operator module
  - functions for common operators, functions to pick items from sequences, read attributes from object (itemgetter, attrgetter)
- functools
  - reduce
  - partial
    - allows partial application of a function
    - takes a callable and keyword and positional arguments to bind
  - partialmethod
    - same as partial but used with methods

Ch. 6 - Design Patterns with First-Class Functions

- design patterns can be drastically modified or obsolete in dynamic languages

Case Study: Refactoring Strategy

- classic pattern performs actions based on the specific instance of a polymorphic type passed and a method called on that instance
- can easily replace with functions when functions are first-class
- skipping material here
- NOTE: simplest way to declare an ABC is to subclass abc.ABC in Python 3.4
  - must use the metaclass = keyword in the class statement from Python 3.0 to 3.3
  - e.g., class Promotion(metaclass = ABCMeta):

- modules are also first-class
    - can encapsulate strategies in modules
- use the inspect module for introspection of modules

Command

- decouple an object that invokes an operation (the Invoker) from the provider object that implements it (the Receiver)
    - each invoker is a menu item in a graphical application, and the receivers are the document being edited or the application itself
    - put a Command object between the Invoker and Receiver
    - Command implements an interface with a single method, execute, which calls some method in the Receiver to perform the desired operation
    - Invoker does not need to know the interface of the Receiver
    - different receivers can be adapted through different Command subclasses
    - Invoker is configured with a concrete command and calls its execute method to operate it
    - may store sequence of commands
    - "Commands are an object-oriented replacement for callbacks"
- to simplify the pattern, can simply give the Invoker a function instead of a Command instance
    - Invoker can just call command() instead of calling command.execute()
- more advanced uses of the Command pattern may need more than a simple callback function
- Python alternatives
    - a callable instance can keep whatever state is necessary, and provide extra methods in addition to `__call__`
    - a closure can be used to hold the internal state of a function between calls

Ch. 7 - Function Decorators and Closures

Decorators 101

- callable that takes another function as argument (the decorated function)
    - performs processing with the decorated function
    - returns function or replaces it with another function or callable object

```
@decorate
def target():
    print('running target()')
```

```
# equivalent to

def target():
    print('running target()')

target = decorate(target)
```

- just syntactic sugar
- usually executed immediately when a module is loaded

When Python Executes Decorators

- decorators run after decorated function is defined - usually at import time
- decorated function only run when explicitly invoked
- import time vs. runtime

Decorator-Enhanced Strategy Pattern

- cleans up over previous examples
    - do not need to use special names
    - makes it easier to modify all decorated functions at once and highlight purpose
    - can define decorated functions within any module

Variable Scope Rules

- variables within a function that are ASSIGNED (important) to and not explicitly declared global are designated as local variables when Python compiles the body of the function
    - can be observed in generated byte code
- Python does not require variables to be declared
    - assumes that a variable assigned in the body of a function is local
- use the global declaration to treat a global variable as global

Closures

- closures are sometimes confused with anonymous functions
    - the use of anonymous functions is what historically led to the definition of functions inside functions
    - closures are important when using nested functions
- a closure is a function with an extended scope that encompasses nonglobal variables referenced in the body of the function but not defined there
    - inconsequential whether function is anonymous or not

    - important aspect is whether function can access nonglobal variables that are defined outside of its body
- free variable - technical term meaning a variable that is not bound in the local scope
    - often used in closures to reference variables in the enclosing function
    - can inspect free variables in `__code__.co_freevars`
- summary
    - a closure is a function that retains the bindings of the free variables that exist when the function is defined
    - free variables can then be used later when the function is invoked and the defining scope is no longer available

The nonlocal Declaration

- `count += 1` is actually `count = count + 1` (when count is a number or immutable type) so the variable becomes local
- can only read to free vars that are immutable types (numbers, strings, tuples, etc)
    - update/assignment implicitly creates a local variable
- `nonlocal` keyword added in Python 3 to tag free variables
    - to handle the lack of nonlocal in Python 2
        - store the variables the inner functions need to change (e.g., count, total) as items or attributes of some mutable object, like a dict or a simple instance, and bind that object to a free variable
        - see third code snippet of PEP 3104

Implementing a Simple Decorator

Decorators in the Standard Library

- three built-in decorators - property, classmethod, staticmethod
- functools.wraps helps to build well-behaved decorators
- functools.lru_cache
    - implements memoization - optimization technique that works by saving the results of previous invocations of an expensive function, avoiding repeat computations on previously used arguments
    - uses an LRU cache
- functools.singledispatch
    - new decorator in Python 3.4
    - transforms a function into a single-dispatch generic function
    - to define a generic function, decorate it with the @singledispatch decorator
        - dispatch happens on the type of the first argument

    - use the register() attribute of the generic function to add overloaded implementations to the function
        - it is a decorator and takes a type parameter and decorating a function implementing the operation for that type
    - decorating a plain function causes it to become a generic function or a group of functions to perform the same operation in different ways, depending on the type of the first argument
    - singledispatch package is available on PyPI as a backport compatible with Python 2.6 to 3.3
    - register the specialized functions to handle ABCs (abstract classes) instead of concrete implementations when possible
        - allows your code to support a greater variety of compatible types
        - allows code to support existing or future classes that are either actual or virtual subclasses of those ABCs
    - can register specialized functions anywhere in the system in any module
    - offers much more functionality (see PEP 443)
    - @singledispatch is not designed to bring Java-style method overloading to Python
        - a single class with many overloaded variations of a method is better than a single function with many if/else blocks
        - both are flawed because they concentrate too much responsibility in a single code unit (class or function)
        - the advantage of @singledispath is support of modular extension
            - each module can register a specialized function for each type it supports
- single dispatch
    - form of generic function dispatch where the implementation is chosen based on the type of a single argument
- generic function
    - function composed of multiple functions implementing the same operation for different types
        - the implementation that should be used during a call is determined by the dispatch algorithm

Stacked Decorators

```
@d1
@d2
def f():
    print('f')

# equivalent to
def f():
    print('f')
```

```
f = d1(d2(f))
```

Parameterized Decorators

- decorator takes decorated function as first arg
- can pass more args by making a decorator factory that takes args and returns a decorator which is then applied to the function in question
- see book code for examples

Ch. 8 - Object References, Mutability, and Recycling

Variables are Not Boxes

- conceptualize Python variables as labels and the label refers to an object
    - as opposed to a box in which objects/values are deposited and updated
- similar to Java references
- good to think of it as the variable is assigned to the object rather than the object is assigned to the variable
- always read the right-hand side first
    - rhs is where the object is created or retrieved
    - after that the variable on the left is bound to the object
    - similar to a label stuck to the object
- an object can have several labels assigned to it (aliasing)

Identity, Equality, and Aliases

- every object has an identity, a type and a value
- an object's identity never changes once it has been created
- `is` operator compares the identity of two objects
- `id()` function returns an integer representing its identity
    - the meaning of an object's ID is implementation-dependent
    - id() returns the memory address of the object in CPython
    - may be something else in another Python interpreter
- ID is guaranteed to be a unique numeric label that will never change during the life of the object
- use of the `id()` function while programming is rare
    - identity checks are most often done with the `is` operator

Choosing Between == and is

- the `==` operator compares the values of (data held by) objects
- `==` appears more frequently than `is` in Python code
- when comparing a variable to a singleton use `is`
- most common case is checking whether a variable is bound to None
  - use `x is None` or `x is not None`
  - `is` operator is faster than `==` because it cannot be overloaded
    - no need for interpreter to find and invoke special methods to evaluate it
    - computing is as simple as comparing two integer IDs
- `a == b` is syntactic sugar for `a.__eq__(b)`
  - `__eq__` method inherited from object compares object IDs, so it produces the same result as `is`
  - most built-in types override `__eq__` with more meaningful implementations that take into account the values of the object attributes
- equality checks may be expensive (collections, nested structures)

The Relative Immutability of Tuples

- tuples hold references to objects
  - if the referenced items may change (if mutable) even if the tuple doesn't
  - immutability of tuples means the specific references contained in the tuple and not the references themselves
  - what never changes in a tuple is the identities of the items they contain

Copies Are Shallow by Default

- easiest copy method for mutable collections is to use the built-in constructor
  - shortcut `l2 = l1[:]` also makes a copy
- produces a shallow copy where outermost container is duplicated but copy is filled with references to same items held by the original container
  - saves memory and doesn't not cause problems with immutable items
  - leads to issues with mutable items where aliases can cause unexpected modification

Deep and Shallow

- copy modules provides `deepcopy` and `copy`
- copy.copy(x) - return a shallow copy of x
- copy.deepcopy(x) - return a deep copy of x
- both raise exception copy.error for module specific errors

- difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances)
  - shallow copy constructs a new compound object and then inserts references into it to the objects found in the original (as able)
  - deep copy constructs a new compound object and recursively inserts copies into it of the objects found in the original
- deep copy problems
  - recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop
  - deep copy may copy too much
- deepcopy() avoids problems by
  - keeping a "memo" dictionary of objects already copied during the current copying pass
  - letting user-defined classes override the copying operation or the set of components copied
- can control the behavior of both copy and deepcopy by implementing the **copy** () and **deepcopy** () special methods as described in the copy module documentation

Function Parameters as References

- Python passes parameters using call by sharing (e.g. pass by reference)
  - each formal parameter of the function gets a copy of each reference in the
  - parameters inside the function become aliases of the actual arguments
- actual arguments passed to functions may behave in different ways depending on the mutability of the arguments
  - with an assignment
    - number is unchanged
    - list is changed
    - tuple is unchanged
- avoid mutable objects as default values to function parameters
  - each default value is evaluated when the function is defined (usually on module load) and then become attributes of the function object
  - changes to the default value are shared across function calls
- carefully consider ownership and the resulting ability to modify a mutable variable passed to a function
  - with lists, for example, instead of using a default empty list, use None
    - first check for None and then create an empty list if None
    - otherwise construct a new list (or deepcopy) using the `list(arg)` ctor
  - avoids the potential to mutate the parameter unknowingly or share a default

del and Garbage Collection

- objects are never explicitly destroyed
  - when they become unreachable they may be garbage-collected
- del statement deletes names, not objects
  - del command may cause an object to be garbage collected
    - if the variable deleted holds the last reference to the object
    - if the object becomes unreachable
- rebinding a variable may also cause the number of references to an object to reach zero
- `__del__` special method does not cause the disposal of the instance
  - should not be called by code
  - invoked by the Python interpreter when the instance is about to be destroyed to give it a chance to release external resources
  - rarely necessary to implement
- proper use of `__del__` is difficult
  - see the `__del__` special method documentation in "Data Model" Python Reference . In CPython, the
- primary algorithm for garbage collection in CPython is reference counting
  - each object keeps count of how many references point to it
  - object is destroyed when refcount reaches zero
    - `__del__` method is called (if defined)
    - memory allocated to the object is destroyed . In CPython 2.0,
- a generational garbage collection algorithm was added in CPython 2.0 to detect groups of objects involved in reference cycles which may be unreachable when all the mutual references are contained within the group
- other implementations of Python have more sophisticated garbage collectors that do not rely on reference counting
  - see "PyPy, Garbage Collection, and a Deadlock"
- REMEMBER: `del` does not delete objects but objects may be deleted after a call to `del` if the refcount drops to 0

Weak References

- references keep an object alive in memory
  - garbage collector disposes of an object when the refcount drops to 0
- sometimes useful to have a reference to an object that does not increment refcount (extending lifetime)
  - common use case is a cache
  - don't want the cached objects to be kept alive just because they are referenced by the cache

- target of a reference is called the referent
- a weak reference does not prevent the referent from being garbage collected
- can use `weak_ref is None` to check if object is alive
- memory management under the hood often contains hidden implicit assignments that create new references
    - _ console variable
    - traceback objects
- weakref.ref class is a low-level interface intended for advanced uses
    - most programs are better served by the use of the weakref collections and finalize
    - consider using WeakKeyDictionary, WeakValueDictionary, WeakSet, and finalize (which use weak references internally) instead
- WeakValueDictionary implements a mutable mapping where the values are weak references to objects
    - corresponding key is automatically removed from WeakValueDictionary when a referred object is garbage collected elsewhere in the program
    - commonly used for caching
- a temporary variable may cause an object to last longer than expected by holding a reference to it
    - usually not a problem with local variables which are destroyed when the function returns
    - be careful with global variables and loops
- WeakKeyDictionary - keys are weak references
    - can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects
    - can be useful with objects that override attribute accesses
- WeakSet - set class that keeps weak references to its elements
    - element will be discarded when no strong reference to it exist any more
    - can be used to build a class that is aware of every one of its instances
        - class attribute with a WeakSet to hold the references to the instances
        - instances would never be garbage collected with a regular set
- not every Python object may be the target, or referent, of a weak reference
    - basic list and dict instances may not be referents
        - plain subclass of either can be
    - a set instance can be a referent
    - user-defined types can be referents
    - int and tuple instances cannot be targets of weak references, even if subclasses of those types are created
    - limitations are implementation details of CPython

Tricks Python Plays with Immutables

- for a tuple t, `t[:]` does not make a copy
  - it returns a reference to the same object
  - `tuple(t)` also returns a reference to the same object
- same behavior can be observed with instances of str, bytes, and frozenset
  - (frozenset is not a sequence, so `fs[:]` does not work if fs is a frozenset)
  - fs.copy() has the same behavior
- sharing of string literals is an optimization technique called interning
  - CPython uses the same technique with small integers to avoid unnecessary duplication of common numbers like 0, –1, and 42
  - CPython does not intern all strings or integers
- NOTE: never depend on str or int interning!!!
  - always use == and not is to compare them for equality
- these tricks save memory and make the interpreter faster and are not important for everyday programming

Ch. 9 - A Pythonic Object

- user-defined types can behave like built-in types without inheritance
  - duck typing - just implement the methods needed

Object Representations

- two methods for getting string representation
  - repr() - developer's view of an object (uses `__repr__`)
  - str() - user's view of an object (uses `__str__`)
- two additional methods for representations
  - `__bytes__` - analogous to str but used to the object represented as a byte string
  - `__format__` - called by `format()` and `str.format()` to get string displays of objects using special formatting codes

Vector Class Redux

- see code for details

An Alternative Constructor

- when supporting encoding as bytes, need to have ability to decode as bytes
  - see `array.array.frombytes`
  - good to use as a classmethod

classmethod Versus staticmethod

- use classmethod to define a method that operates on the class and not on instances
  - changes the way the method is called
  - receives the class as the first argument instead of an instance
  - most commonly used for alternative constructors
    - often returns an instance using the class argument
  - first parameter of a class method should be named cls (by convention)
- staticmethod decorator changes a method so that it receives no special first argument
  - like a plain function that happens to live in a class body
- compelling use cases for staticmethod are hard to find
  - just define functions in the module

Formatted Displays

- `format()` built-in and `str.format()` delegate formatting to each type by calling `.__format__(format_spec)`
  - format_spec is a formatting specifier, either
    - second argument in format(my_obj, format_spec)
    - whatever appears after the colon in a replacement field delimited with {} inside a format string used with str.format()
- see Format Spec Mini-Languages and Format String Syntax in docs

A Hashable Vector2d

- implement `__hash__` and `__eq__`
  - if an object implements a custom `__eq__` that takes into account its internal state it may be hashable only if all its attributes are immutable
  - must make vector instances immutable
- restricting access to attributes
  - NOTE: use two leading underscores to make an attribute private!!!
  - define a function with the attribute name (without underscores) and decorate with the @property decorator - the most simple case is just returning the private attribute
  - for methods that only read the value, prefer accessing the public property

```
self.__x = val

@property
def x(self):
    return self.__x
```

- define `__hash__` (can use the hash built-in and XOR the components of an object)
    - see docs for `__hash__` special method

Private and "Protected" Attributes in Python

- no real protection, just a method for preventing overwriting of "private" attributes in a subclass
- name an instance attribute with two leading underscores and zero or at most one trailing underscore
    - Python stores the name in the instance `__dict__` prefixed with a leading underscore and the class name (`_Class__attribute`)
    - name mangling in Python
        - designed to prevent accidental access, but not designed for security
- some prefer to use just one underscore prefix to "protect" attributes (e.g. `self._x`)
    - some believe accidental attribute clobbering should be addressed by naming conventions rather than double-underscore mangling
    - some explicitly state not to use double-underscore for private
- single underscore prefix has no special meaning to the Python interpreter when used in attribute names
    - strong convention among Python programmers that these attributes should not be accessed outside of the class
- single underscore prefixed attributes are sometimes called "protected", some call them "private"
- NOTE: no way to really make attributes private and immutable

Saving Space with the `__slots__` Class Attribute

- Python stores instance attributes in a per-instance dict named `__dict__`
    - dictionaries have a significant memory overhead
- `__slots__` class attribute can save a lot of memory by letting the interpreter store the instance attributes in a tuple instead of a dict
    - good for dealing with many objects
- NOTE: `__slots__` attribute inherited from a superclass has no effect
    - Python only takes into account `__slots__` attributes defined in each class

- create a class attribute with the name `__slots__` and assign it an iterable of str with identifiers for the instance attributes
  - good to use a tuple because it conveys the message that the `__slots__` definition cannot change By defining **slots** in the class, you are telling the interpreter: "
- effectively tells interpreter that the specified attributes are all the instance attributes in this class
- stored in a tuple-like structure in each instance
  - avoids the memory overhead of the per-instance `__dict__`
- NOTE: use numpy arrays if handling millions of objects with numeric data
  - memory-efficient
  - have highly optimized functions for numeric processing
- class instances will not be allowed to have any other attributes when **slots** is specified in a class
  - considered bad form to use **slots** just to prevent users of your class from creating new attributes in the instances if they want to
  - use for optimization, not for programmer restraint
  - can add the `__dict__` name to the `__slots__`
    - instances will keep attributes named in **slots** in the per-instance tuple
    - will also support dynamically created attributes which will be stored in the usual `__dict__`
    - may defeat the purpose
- the `__weakref__` attribute is necessary for an object to support weak references
  - present by default in instances of user-defined classes
  - need to include `__weakref__` in `__slots__` if instances may need to be targets of weak references
- problems with `__slots__`
  - must remember to redeclare `__slots__` in each subclass
    - inherited attributes are ignored by the interpreter
  - instances will only be able to have the attributes listed in `__slots__`
    - unless including `__dict__` in `__slots__` which may negate the memory savings
  - instances cannot be targets of weak references unless `__weakref__` is included in `__slots__`
- mainly beneficial and to be considered for working with a large number of instances

Overriding Class Attributes

- class attributes can be used as default values for instance attributes
- writing to an instance attribute that does not exist creates a new instance attribute
  - a class attribute by the same name is untouched

    - any code reading that attribute from that point on will read the instance attribute rather than the class attribute (shadowing the class attribute name)
    - allows for customizing individual instances by overriding a class attribute
- to change a class attribute, set it on the class specifically rather than through an instance
- common practice to inherit from a class just to override a class attribute (see Django)
- can read the class name rather than hardcoding with

```
class_name = type(self).__name__
```

Ch. 10 - Sequence Hacking, Hashing, and Slicing

Vector: A User-Defined Sequence Type

- using composition over inheritance
    - provide access to an array of floats to create an immutable flat sequence type

Vector Take #1: Vector2d Compatible

- NOTE: for > 6 items, repr abbreviates with '...'
    - use reprlib for more
- repr should never raise an exception because of it's use in debugging
- see section for code

Protocols and Duck Typing

- a protocol is an informal interface defined only in documentation and not in code (in Python)
    - whether a class is a subclass of the protocol class is irrelevant - all that matters is that it provides the necessary methods
- duck typing (after Alex Martelli's post)
    - behaving like a sequence is all that matters, i.e. implementing the sequence protocol
    - does not need to be declared anywhere in the code
- can often get away with implementing just part of a protocol (informal and unenforced)
    - to support iteration, only `__getitem__` is required

Vector Take #2: A Sliceable Sequence

- see code for details

- delegates `__len__` and `__getitem__` to the `self._components` array
    - need to modify `__getitem__` to ensure slices produce Vector items rather than array items
- must know slice object to better understand `__getitem__`
    - attributes start, stop, and step, and an indices method
    - `S.indices(len) -> (start, stop, stride)`
        - assuming a sequence of length len, calculate the start and stop indices, and the stride length of the extended slice described by S
        - out of bounds indices are clipped in a manner consistent with the handling of normal slices
    - indices basically normalizes the values passed to slices
- improved `__getitem__` body

```
cls = type(self)
if isinstance(index, slice):
    return cls(self._components[ index])
elif isinstance(index, numbers.Integral):
    return self._components[index]
else:
    msg = '{cls.__name__} indices must be integers'
    raise TypeError(msg.format(cls=cls))
```

- NOTE: excessive use of isinstance may be a sign of bad OO design
    - using ABCs in isinstance tests makes an API more flexible and future-proof
    - no ABC for slice in the Python 3.4 standard library

Vector Take #3: Dynamic Attribute Access

- the `__getattr__` method is invoked by the interpreter when attribute lookup fails
    - a call to `obj.x` causes the interpreter to check if the instance has an attribute named x
    - if not, the search goes to the `class(obj.__class__)`
    - then it goes up the inheritance graph
    - if the x attribute is not found, then the `__getattr__` method defined in the class of obj is called with self and the name of the attribute as a string (e.g., 'x')
- NOTE: `__getattr__` is only called as a fallback
    - if an attribute is dynamically assigned to an instance, the lookup described above will not occur
- can use `__setattr__` to prevent assignment of undesired names, or generally control assignment to attributes
- NOTE: the `super()` function a way to access methods of superclasses dynamically
    - necessary in a dynamic language that supports multiple inheritance
    - used to delegate some task from a method in a subclass to a method in a superclass

- common to need to implement `__setattr__` when `__getattr__` is required

Vector Take #4: Hashing and a Faster ==

- using functools.reduce to apply hash and XOR to each component of the array
  - can be replaced by sum in many cases

```
hashes = (hash(x) for x in self._components)
return functools.reduce(operator.xor, hashes, 0)
```

- good practice to provide the third argument, reduce(function, iterable, initializer), to prevent a TypeError on an empty sequence with no initial value
  - initializer is the value returned if the sequence is empty and is used as the first argument in the reducing loop
  - should be the identity value of the operation
  - for +, |, ^ the initializer should be 0
  - for *, & it should be 1
- perfect example of map/reduce

```
hashes = map(hash, self._components)
return functools.reduce(operator.xor, hashes)
```

- NOTE: map is less efficient in Python 2 because it builds a new list
  - Python 3 builds a lazy generator
- can also replace `__eq__`

```
return len(self) == len(other) and all(a == b for a, b in zip(self, other))
```

Vector Take #5: Formatting

- NOTE: when extending the Format Specification Mini-Language it's best to avoid reusing format codes supported by built-in types
- see code for support of hyperspherical coordinates

Ch. 11 - Interfaces: From Protocols to ABCs

Interfaces and Protocols in Python Culture

- every class has an interface
    - the set public attributes (methods or data attributes) implemented or inherited by the class
    - includes special methods, like `__getitem__` or `__add__`
    - protected and private attributes are not part of an interface by definition
        - even if "protected" is merely a naming convention (the single leading underscore)
        - even if private attributes are easily accessed
    - possible and even OK to have public data attributes as part of the interface of an object
        - if necessary a data attribute can be turned into a property implementing getter/ setter logic without breaking client code
- complementary definition of interface
    - the subset of an object's public methods that enable it to play a specific role in the system
- interface as a set of methods to fulfill a role is what == procotol in Smalltalk
- protocols are independent of inheritance
    - a class may implement several protocols
    - enables instances to fulfill several roles
- protocols are informal interfaces and are therefore not enforced like formal interfaces
    - may be partially implemented in a particular class

Python Digs Sequences

- see code

Monkey-Patching to Implement a Protocol at Runtime

- following established protocols improves chances of working with existing library functions etc
- monkey-patching - changing code at runtime
- shows example of adding methods to a class at runtime to allow it to work with random.shuffle
- alsow highlights that protocols are dynamic

Alex Martelli's Waterfowl

- Alex Martelli - Waterfowl and ABCs
    - duck typing
        - ignoring an object's actual type
        - ensure that the object implements the method names, signatures, and semantics required for its intended use
    - mostly boils down to avoiding the use of isinstance to check the object's type in Python
        - also worse approach of checking whether `type(foo) is bar`
        - inhibits even the simplest forms of inheritance
    - alternate approach to duck typing has evolved over time
    - the presence of two identical methods on a type doesn't imply interchangeability
        - programmer needs to be able to assert something to be assured of interchangeability
    - duck typing -> goose typing
    - major advantage of ABCs
        - the register class method lets end-user code "declare" that a certain class becomes a "virtual" subclass of an ABC
        - registered class must meet the ABC's method name and signature requirements
        - more importantly must meet the underlying semantic contract of the signature requirements
        - does not have to have been developed with any awareness of the ABC (i.e. inherit from it)
        - breaks rigidity and strong coupling that make inheritance something to use with much more caution than typically practiced by most OOP
    - in some cases, classes do not even need to be registered as a subclass to be recognized
    - use `isinstance(instance, ABC)`
    - don't define custom ABCs in production code
- inheriting from an ABC is more than implementing the required methods
    - also a declaration of intent by the developer
    - intent can also be made explicit through registering a virtual subclass
- use of isinstance and issubclass becomes more acceptable to test against ABCs
    - these functions previously worked against duck typing
    - if a component does not implement an ABC by subclassing, it can be registered after the fact so it passes those explicit type checks . However, even with ABCs, you should beware that

- excessive use of isinstance checks may be a code smell even with ABCs
    - should use polymorphism for that
    - design classes so that the interpreter dispatches calls to the proper methods
    - exceptions
        - some Python APIs accept a single str or a sequence of str items
            - wrap the str in a list to ease processing Because str is a sequence type,
        - the simplest way to distinguish the string from any other immutable sequence is to do an explicit `isinstance(x, str)` (since it is a sequence type)
    - usually OK to perform an insinstance check against an ABC if an API contract must be enforces
- duck typing is often simpler and more flexible than type checks outside of frameworks
- do not overuse ABCs
    - ABCs are meant to encapsulate very general concepts, abstractions, introduced by a framework
    - most programmers most likely don't need to write any new ABCs
        - use existing ones correctly

Subclassing an ABC

- interpreter does not check for the implementation of the abstract methods at import time
    - only checked at runtime when an implementing instance is instantiated
    - if any abstract method is not implemented a TypeError exception is thrown
- concrete methods in each collections.abc ABC are implemented in terms of the public interface of the class
    - work without any knowledge of the internal structure of instances
- NOTE: when coding a concrete subclass it may be possible to override methods inherited from ABCs with more efficient implementations
    - `__contains__` works by doing a full scan of the sequence
        - can speed a sorted concrete sequence using binary search with bisect

ABCs in the Standard Library

- ABCs have been available since 2.6
    - most in collections.abc module
    - numbers and io packages
- two modules named abc in the standard library
    - in Python 3.4 implemented outside of the collections package to reduce load time

  - other abc module is just abc (i.e., Lib/ abc.py)
    - defines abc.ABC class
    - depended on by every ABC but not needed to extend ABCs
- see figure 11-3 or the table in the docs for inheritance structure in ABC
- much multiple inheritance, but mostly for mixin methods
- Iterable, Container, and Sized
  - every collection should either inherit from these ABCs or at least implement compatible protocols
  - Iterable supports iteration with `__iter__`
  - Container supports the in operator with `__contains__`
  - Sized supports `len()` with `__len__`
- Sequence, Mapping, and Set
  - main immutable collection types
  - each has a mutable subclass
- MappingView
  - objects returned from the mapping methods `.items()` , `.keys()` , and `.values()` inherit from ItemsView, ValuesView, and ValuesView in Python 3
  - first two also inherit the interface of Set
- Callable and Hashable
  - not limited to collections
  - collections.abc was the first package to define ABCs in the standard library
  - rare to see subclasses of either Callable or Hashable
  - main use is to support the `insinstance` built-in as a safe way of determining whether an object is callable or hashable
- Iterator
  - iterator subclasses Iterable
- numbers packages
  - Number
  - Complex
  - Real
  - Rational
  - Float
  - can use each with `isinstance`

- decimal.Decimal is not registered as a virtual subclass of numbers.Real
    - need protection from accidental mixing of decimals with other less precise numeric types (i.e. floats) when the precision of Decimal is require

Defining and Using an ABC

- common to use in frameworks

- example class is designed designed to pick items at random from a finite set, without repeating, until the set is exhausted

- abstract methods

    - `.load():` - put items into the container
    - `.pick():` - remove one item at random from the container, returning it

- concrete methods

    - `.loaded():` - return True if there is at least one item in the container
    - `.inspect():` - return a sorted tuple built from the items currently in the container, without changing its contents (its internal ordering is not preserved)

- mark an abstract method with @abc.abstractmethod and generally leave the body empty (but add a docstring)

- NOTE: an abstract method can have an implementation

    - subclasses will still be forced to override it
    - will be able to invoke the abstract method with super()
        - adds functionality to it instead of implementing from scratch
    - see the abc module documentation for details on @abstractmethod usage

- to declare an ABC, subclass abc.ABC or any other ABC

    - abc.ABC class is new in Python 3.4
    - for earlier versions of Python use the metaclass=keyword in the class statement and point to abc.ABCMeta `class Tombola(metaclass=abc.ABCMeta):`

- metaclass=keyword argument was introduced in Python 3

- use the **metaclass** class attribute in Python 2

```
class Tombola(object):
    __metaclass__ = abc.ABCMeta
```

- a metaclass is a special kind of class explained later

- abc module defines the following decorators
  - @abstractmethod
  - @abstractclassmethod
  - @abstractstaticmethod
  - @abstractproperty
  - last three are deprecated since Python 3.3
    - possible to stack decorators on top of @abstractmethod
    - the preferred way to declare an abstract class method is

```
@classmethod
@abc.abstractmethod
def an_abstract_classmethod(cls, ...):
    pass
```

- NOTE: order of stacked function decorators matters, check docs

- see code for details of subclassing the custom ABC

- with goose typing programmer can register a class as a virtual subclass of an ABC even if it does not inherit from it

  - declares contract that the class faithfully implements the interface defined in the ABC
  - interpreter assumes this is true without checking
  - if not implemented, runtime exceptions will occur

- done by calling a register method on the ABC

  - registered class then becomes a virtual subclass of the ABC
    - will be recognized by functions like issubclass and isinstance
    - will not inherit any methods or attributes from the ABC

- NOTE: virtual subclasses do not inherit from their registered ABCs

  - they are not checked for conformance to the ABC interface at any time
  - up to the subclass to actually implement all the methods needed to avoid runtime errors

- register method is usually invoked as a plain function

  - can also used as a decorator `@ABCClass.register`

- `issubclass` and `isinstance` work but `__mro__` does not show the ABC as a class that is inherited from

How the Tombola Subclasses Were Tested

- class attributes that allow introspection of a class hierarchy
    - `__subclasses__()`
        - method that returns a list of the immediate subclasses of the class
            - does not include virtual subclasses
    - `_abc_registry`
        - data attribute that is only available in ABCs
        - bound to a WeakSet with weak references to registered virtual subclasses of the abstract class
- see code for details

Usage of register in Practice

- `register` had to be called as plain function prior to Python 3.3
    - often still called as a function e.g. `Sequence.register(tuple)`

Geese Can Behave as Ducks

- class can be recognized as a virtual subclass of an ABC even without registration Struggle is considered a subclass of abc.Sized by the
- issubclass and isinstance functions consider example class to be a subclass of the ABC because the ABC implements a special class method named `__subclasshook__`
    - for Sized, checks if `__len__` is in the `__dict__` for all of the inherited classes in `__mro__` of the class being checked
- probably not a good idea to implement `__subclasshook__` in your own ABCs

Ch. 12 - Inheritance: For Good or For Worse

Subclassing Built-In Types Is Tricky

- not possible to subclass built-in types before Python 2.2.
- can now be done
    - code of the built-ins, written in C, does not call special methods overridden by user-defined classes
    - CPython has no rule for when overridden method of subclasses of built-in types get implicitly called or not
        - methods are never called by other built-in methods of the same object

- violates basic rule of OO
    - method search should always start from base class of target instance and propagate upwards
- problem applies only to method delegation within the C language implementation of the built-in types
    - only affects user-defined classes derived directly from those types
    - subclassing from a class coded in Python will not cause the same problems

Multiple Inheritance and Method Resolution Order

- consider diamond inheritance classes in multiple inheritance
- can explicitly call superclass methods

```
i.method()
P.method(i)
```

- Python follows a specific order when traversing the inheritance graph
    - called MRO - Method Resolution Order
    - classes have an attribute called `__mro__` holding a tuple of references to the superclasses in MRO order
        - current class all the way to the object class
- recommended way to delegate method calls to superclasses is the super() built-in function
    - easier to use in Python 3
- possible to bypass the MRO and invoke a method on a superclass directly
    - must pass self explicitly when calling an instance method directly on a class
        - accessing an unbound method
- safest and most future-proof to call super
    - especially when using frameworks and class hierarchies that are externally controlled
- MRO is computed using an algorithm called C3
    - "The Python 2.3 Method Resolution Order" is the canonical paper
    - not necessary to understand unless making strong use of multiple inheritance

Multiple Inheritance in the Real World

- Adapter pattern uses MI
- collections.abc uses MI but more for interfaces
- Tkinter GUI toolkit uses extensively due to widget-based approach
- see book for descriptions of classes and hierarchy

Coping with Multiple Inheritance

1. distinguish interface inheritance from implementation inheritance
   - useful to keep the reasons why subclassing is done straight when using MI
     - inheritance of interface creates a subtype, implying an "is-a" relationship
     - inheritance of implementation avoids code duplication by reuse
   - both uses are often simultaneous
     - make intent clear when possible
   - code reuse is an implementation detail and can often be replaced by composition and delegation
   - interface inheritance is the backbone of a framework
2. make interfaces explicit with abcs
   - if a class is designed to define an interface it should be an explicit ABC in modern Python
     - subclass abc.ABC or another ABC
3. use mixins for code reuse
   - a class is designed to provide method implementations for reuse by multiple unrelated subclasses should be an explicit mixin class if it doesn't imply an "is-a" relationship
   - a mixin does not define a new type (conceptually)
     - bundles methods for reuse
   - never instantiate
   - concrete classes should not only inherit from a mixin
   - a single mixin should provide a single specific behavior with a few closely related methods
4. make mixins explicit by naming
   - no formal method to define a mixin
     - highly recommended to name with a Mixin suffix
5. an abc may also be a mixin; the reverse is not true
   - when an ABC can implement concrete methods it can work as a mixin also
   - an ABC defines a type
     - mixin does not
   - an ABC can be the sole base class of any other class
     - mixin should not be (except in special cases)
   - a restriction applied to ABCs and not mixins
     - concrete methods implemented in an ABC should only collaborate with methods of the same ABC and its superclasses
6. don't subclass from more than one concrete class concrete
   - classes should have zero or at most one concrete superclass
   - all but one of the superclasses of a concrete class should be ABCs or mixins

7. provide aggregate classes to users
    - provide a class that brings classes together in a sensible way if some combination of ABCs or mixins is useful to client code
    - aggregate class
8. favor object composition over class inheritance
    - best advice
    - easy to overuse inheritance
    - hierarchy appeals to a sense of order
    - favoring composition leads to more flexible designs
    - in many cases, a class that

- see text for issues with Tkinter

A Modern Example: Mixins in Django Generic Views

- see text for description

Ch. 13 - Operator Overloading: Doing It Right

Operator Overloading 101

- can be abused, but can also lead to clean programming constructs
- limitations in Python
    - cannot overload operators for the built-in types
    - cannot create new operators, only overload existing ones
    - some operators can't be overloaded - is, and, or, not (but the bitwise &, |, ~, can)

Unary Operators

- `-(__neg__)`
    - arithmetic unary negation
- `+(__pos__)`
    - arithmetic unary plus
    - usually x == +x
    - cases not true
- `~(__invert__)`
    - bitwise inverse of an integer, defined as ~x == -(x + 1)
- abs() built-in function is listed as a unary operator
    - `__abs__` special method

- to support, implement the appropriate special method, which will receive just one argument - self
  - use logic that makes sense for the class
  - fundamental rule of operators - always return a new object
    - do not modify self
    - create and return a new instance of a suitable type
- cases where x and +x are not equal

Everybody expects that x = = + x, and that is true almost all the time in Python, but I found two cases in the standard library where x != + x. The first case involves the - decimal.Decimal class - can have x != +x if x is a Decimal instance created in an arithmetic context and +x is then evaluated in a context with different settings - collections.Counter documentation - Counter class implements infix + to add the tallies from two Counter instances - Counter addition discards from the result any item with a negative or zero count - prefix + is a shortcut for adding an empty Counter - produces a new Counter preserving only the tallies that are greater than zero

Overloading + for Vector Addition

- for Sequence types, + and * should be used for concatenation and repetition
- see book for code examples
- NOTE: only augmented assignment operators should modify self
  - all others should return a new object
- NOTE: do not confuse NotImplemented with NotImplementedError
  - NotImplemented is a special singleton value that an infix operator special method should return to tell the interpreter it cannot handle a given operand
  - NotImplementedError is an exception that stub methods in abstract classes raise to warn that they must be overwritten by subclasses
- NOTE: an infix operator that raises an exception aborts the operator dispatch algorithm
  - for TypeError it is often better to catch it and return NotImplemented
  - allows the interpreter to try calling the reversed operator method

Overloading * for Scalar Multiplication

- see book for code examples
- see table 13-1 for infix operator method names
- Python 3.5 adds a dedicated infix matrix multiplication operator

Rich Comparison Operators

- operators ==, !=, >, <, >=, <= differ in two aspects
  - same set of methods are used in forward and reverse operator calls

  - both the forward and reverse calls invoke `__eq__` for ==, only swapping arguments
  - a forward call to `__gt__` is followed by a reverse call to `__lt__` with the swapped arguments
  - for == and !=, if the reverse call fails, Python compares the object IDs instead of raising TypeError
- see table 13-2 for rich comparison operators
- NOTE: the fallback step for all comparison operators changed from Python 2
  - Python 3 returns the negated result of `__eq__` for `__ne__`
  - Python 3 raises TypeError for the ordering comparison operators
  - in Python 2 ordering comparisons produced weird results taking into account object types and IDs in some arbitrary way
    - raising TypeError is an improvement because it makes more sense for comparing something like a tuple and an int
- Python 3 documentation bug
  - the rich comparison method documentation states that the truth of x == y does not imply that x! = y is false
  - when defining `__eq__()` , one should also define `__ne__()` so that the operators will behave as expected
    - true for Python 2
    - not good in Python 3
      - a useful default `__ne__` implementation is inherited from the object class
      - rarely necessary to override it

Augmented Assignment Operators

- if the in-place operators are not implemented by a class the augmented assignment operators are just syntactic sugar
  - a += b is evaluated exactly as a = a + b
  - expected behavior for immutable types
  - if `__add__` is implemented then += will work with no additional code
  - if an in-place operator method such as `__iadd__` is implemented, that method is called to compute the result of a += b
- expected to change the lefthand operand in place
  - not create a new object as the result
- NOTE: in-place special methods should never be implemented for immutable types
- NOTE: can observe contrasting behavior of + and += by looking at how the list built-in works
  - can only concatenate one list to another list with infix +
  - can extend the lefthand list with items from any iterable on the righthand side of operator +=
    - consistent with how the list.extend() method works

- `__add__`
    - result is produced by calling the constructor to build a new instance
- `__iadd__`
    - result is produced by returning self after it has been modified . To wrap up this example, a final observation on Example 13-18: by design,
- `__radd__` may not be necessary
    - the forward method `__add__` will only deal with righthand operands of the same type
- NOTE: if a forward infix operator method (e.g., `__mul__`) is designed to work only with operands of the same type as self it is useless to implement the corresponding reverse method (e.g., `__rmul__`)
    - by definition reverse method will only be invoked when dealing with an operand of a different type

Ch. 14 - Iterables, Iterators, and Generators

- abstracting away the Iterator pattern required changing the language
    - the yield keyword was added in Python 2.2
        - allows the construction of generators, which work as iterators
- NOTE: every generator is an iterator
    - generators fully implement the iterator interface
        - an iterator retrieves items from a collection
        - a generator can produce items
            - good for producing an infinite series of numbers that cannot be stored in a collection
- the Python community treats iterator and generator as synonyms most of the time
- Python 3 prefers generators where possible
    - range() is a generator (must use `list(range(n))`)
- every collection in Python is iterable
    - iterators are used to support
        - for loops
        - collection types construction and extension
        - looping over text files line by line
        - list, dict, and set comprehensions
        - tuple unpacking
        - unpacking actual parameters with * in function calls

Sentence Take #1: A Sequence of Words

- see book for code of Sentence class
- sequences are iterable because of the iter function
    - interpreter automatically calls `iter(x)` to iterate over an object

- the iter built-in
    - checks whether the object implements `__iter__`
    - calls it to obtain an iterator
    - if `__iter__` is not implemented but `__getitem__` is
        - Python creates an iterator that attempts to fetch items in order from index 0
    - Python raises TypeError if that fails
- any Python sequence is iterable because they implement `__getitem__`
    - standard sequences also implement `__iter__`
    - special handling of `__getitem__` exists for backward compatibility reasons and may be gone in the future (although it is not deprecated as I write this). As mentioned in "Python Digs Sequences", this is an extreme form of duck typing:
- an object is considered iterable when it implements the special method `__iter__` and/or when it implements `__getitem__` (assuming `__getitem__` accepts int keys starting from 0)
- definition for an iterable is simpler but not as flexible in goose typing
    - an object is considered iterable if it implements the `__iter__` method
- NOTE: check for iterables by calling `iter(x)` and handle TypeError
    - prefer to just try to iterate and handle TypeError

Iterables Versus Iterators

- iterable
    - any object from which the iter built-in function can obtain an iterator
    - objects implementing an `__iter__` method returning an iterator are iterable
    - sequences are always iterable
    - objects implementing a `__getitem__` method that takes 0-based indexes are also iterable
- Python obtains iterators from iterables
- the general process of the `for` loop
    - build an iterator from the iterable
    - repeatedly call next on the iterator to obtain the next item
    - iterator raises `StopIteration` when there are no further items
    - discard iterator object
    - exit the loop
- StopIteration signals that the iterator is exhausted
    - handled internally in for loops and other iteration contexts like list comprehensions, tuple unpacking, etc.
- standard interface for an iterator has two methods
    - `__next__`
        - returns the next available item, raising StopIteration when there are no more items

    - `__iter__`
        - returns self
        - allows iterators to be used where an iterable is expected
- formalized in the collections.abc.Iterator ABC
    - defines `__next__` abstract method
    - subclasses Iterable where the abstract `__iter__` method is defined
- Iterable and Iterator ABCs
    - a concrete `Iterable.__iter__` should return a new Iterator instance
    - a concrete Iterator must implement `__next__`
    - `Iterator.__iter__` method just returns the instance itself
    - Iterator implements **iter** by returning self
        - allows an iterator to be used wherever an iterable is required
- return the next item from the iterator
- the Iterator ABC abstract method is `it.__next__()` in Python 3 and `it.next()` in Python 2
    - avoid calling special methods directly
    - use `next(it)` which does the right thing in Python 2 and 3
- code comments suggest to use hasattr to check for both `__iter__` and `__next__`
    - the `__subclasshook__` method of the abc.Iterator ABC does this
    - the best way to check if an object is an iterator is to call `isinstance(x, abc.Iterator)`
    - test works even if the class is not a real or virtual subclass of Iterator due to `Iterator.__subclasshook__`
- the only methods required of an iterator are `__next__` and `__iter__`
- no way to check whether there are remaining items other than to call next() and catch StopInteration
    - not possible to reset an iterator
    - need to call iter() on the iterable again rather than on the iterator
- iterator definition
    - any object that implements the `__next__` no-argument method that returns the next item in a series or raises `StopIteration` when there are no more items
    - Python iterators also implement the `__iter__` method so they are iterable as well

Sentence Take #2: A Classic Iterator

- uses `__iter__` special method, which is how the iterator design pattern is described in GOF
- a common cause of errors in building iterables and iterators is to confuse the two
    - iterables have an `__iter__` method that instantiates a new iterator every time

    - iterators implement a `__next__` method that returns individual items and an `__iter__` method that returns self
- iterators are iterable -> iterables are not iterators
- it is a bad idea to make the sentence class an iterator
    - common anti-pattern
- GOF - use the Iterator pattern
    - to access an aggregate object's contents without exposing its internal representation
    - to support multiple traversals of aggregate objects
    - to provide a uniform interface for traversing different aggregate structure i.e. polymorphic iteration
- must be possible to obtain multiple independent iterators from the same iterable instance to support multiple traversals
    - each iterator must keep its own internal state
    - proper implementation of the pattern requires each call to `iter(it)` to create a new independent iterator
- NOTE: an iterable should never act as an iterator over itself
    - iterables must implement `__iter__` but not `__next__`
    - iterators should always be iterable
        - `__iter__` should just return self

Sentence #3: A Generator Function

- How a Generator Function Works
- any Python function that has the yield keyword in its body is a generator function
    - a function which, when called, returns a generator object
    - i.e. a generator factory
- NOTE: the only syntax distinguishing a plain function from a generator function is the fact that the latter has a yield keyword somewhere in its body
- the body of a generator function usually has a loop (not necessarily)
    - can have multiple calls to yield
- the written function is a function object
- the invoked function returns a generator object
- generators are iterators that produce the values of the expressions passed to yield
- can assign returned value (an iterator, essentiall) to a variable
    - calling `next()` on this variable will produce the next yielded result
    - generator object raises a StopIteration when the body of the function completes
- a generator function builds a generator object that wraps the body of the function
    - invoking `next(...)` on the generator object causes execution to advance to the next yield in the function body
        - evaluates to the value yielded when the function body is suspended

    - when the function body returns, the enclosing generator object raises StopIteration
- NOTE: often more clear to say a generator yields or produces values rather than returning values

Sentence Take #4: A Lazy Implementation

- the Iterator interface is designed to be lazy
    - `next(it)` produces one item at a time
- eager is the opposite
    - lazy evaluation and eager evaluation are technical terms in programming language theory
- building a list is eager
- building a full list to iterate over does more work than necessary when building
- often a method for remodeling something in a lazy way in Python 3
    - re.finditer function is a lazy version of re.findall and saves a lot of memory

Sentence Take #5: A Generator Expression

- simple generator functions can often be replace by generator expressions
- genexps can be understood as a lazy list comprehension
    - returns a generator that will lazily produce items on demand rather than eagerly building a list
- list comprehension == factory of lists
- generator expression == factory of generators.

Generator Expressions: When to Use Them

- list comprehensions generally work were generator expressions can be used where
    - listcomps use more memory to store intermediate list values
- generator functions are more flexible than generator expressions
    - can code complex logic
    - can use as coroutines
- genexp is easier to use and read
- rule of thumb: prefer a generator function when a generator expression spans more than a couple of lines

Another Example: Arithmetic Progression Generator

- see code for ArithmeticProgression

- itertools has many generator functions that can be combined
    - count can count from 0 or take a start and step
        - infinite - do not call `list(count())`
    - takewhile produces a generator that consumers another generator and stops when a given predicate evaluates to False

Generator Functions in the Standard Library

- see text for lists of generator functions

New Syntax in Python 3.3: yield from

- `yield from` can yield each element from a given iterable without manually looping through the iterable

Iterable Reducing Functions

- reducing == folding == accumulating
- see table 14-6 for reducing functions
- all can be implemented with `functools.reduce` but functions like `all` and `any` short-circuit
- sorted builds an actual list whereas reversed is a generator function

A Closer Look at the iter Function

- iter can be called with two arguments to create an iterator from a regular function or any callable object
    - first argument must be a callable to be invoked repeatedly (with no arguments) to yield values
    - second argument is a marker value which, when returned by the callable, causes the iterator to raise StopIteration instead of yielding the sentinel
    - must rebuild the iterator by invoking `iter(...)` again to start over
- see the iter function built-in documentation

Case Study: Generators in a Database Conversion Utility

- see isis2json.py script

Generators as Coroutines

- PEP 342 — Coroutines via Enhanced Generators was implemented in Python 2.5
  - added extra methods and functionality to generator objects (`.send()`)
- `.send()` causes the generator to advance to the next yield
  - also allows the client using the generator to send data into it
    - any argument passed to .send() becomes the value of the corresponding yield expression inside the generator function body
  - `.send()` allows two-way data exchange between the client code and the generator
  - `.__next__()` which only lets the client receive data from the generator
- changes the nature of generators
  - become coroutines when used in this way
- differences from David Beazley
  - generators produce data for iteration
  - coroutines are consumers of data
  - don't mix the two concepts (for simplicity)
  - coroutines are not related to iteration
  - use case for having yield produce a value in a coroutine
    - not tied to iteration

Ch. 15 - Context Managers and else Blocks

Do This, Then That: else Blocks Beyond if

- else clause can be used in if statements and in for, while, and try statements
- different semantics of for/else, while/else, and try/else than if/else Initially the word else actually hindered my understanding of these features, but eventually I got used to it. Here are the
- rules
  - for
    - else block will run only if and when the for loop runs to completion (i.e., not if the for is aborted with a break)
  - while
    - else block will run only if and when the while loop exits because the condition became falsy (i.e., not when the while is aborted with a break)
  - try
    - else block will only run if no exception is raised in the try block
    - exceptions in the else clause are not handled by the preceding except clauses

- else clause (in all cases) is also skipped if an exception or a return, break, or continue statement causes control to jump out of the main block of the compound statement
- NOTE: then may be a better keyword replacement for else in this case
- using else with these statements often makes the code easier to read and saves the trouble of setting up control flags or adding extra if statements

Context Managers and with Blocks

- context manager objects exist to control a with statement
- the with statement was designed to simplify the try/finally pattern
  - guarantees that some operation is performed after a block of code even if the block is aborted because of an exception, a return or sys.exit() call
- a finally clause usually releases a critical resource or restores some previous state that was temporarily changed
- the context manager protocol consists of the `__enter__` and `__exit__` methods
  - `__enter__` is invoked on the context manager object at the start of the with
  - a call to `__exit__` on the context manager object at the end of the with block plays the role of the finally clause
- the context manager object is the result of evaluating the expression after with
  - the value bound to the target variable (in the as clause) is the result of calling `__enter__` on the context manager object . It just happens that in Example 15-1, the open() function returns an instance of TextIOWrapper, and
- the `open()` returns a context manager and its `__enter__` method returns self
- the `__enter__` method may also return some other object instead of the context manager
- the `__exit__` method is invoked on the context manager object when control flow exits the with block
  - not on whatever is returned by `__enter__`
- the as clause of the with statement is optional (depending on the context manager)
- NOTE: use contextlib.redirect_stdout to redirect stdout
- three arguments passed to **exit**
  - exc_type - exception class

    - exc_value - exception instance (parameters passed to the exception constructor can be found in exc_value.args) traceback - traceback object

The contextlib Utilities

- see contextlib — Utilities for with-statement contexts in the docs for info on creating context managers
- closing
    - a function to build context managers out of objects that provide a close() method but don't implement the `__enter__` / `__exit__` protocol
- suppress
    - a context manager to temporarily ignore specified exceptions
- @contextmanager
    - a decorator that lets you build a context manager from a simple generator function, instead of creating a class and implementing the protocol
- ContextDecorator
    - a base class for defining class-based context managers that can also be used as function decorators, running the entire function within a managed context
- ExitStack
    - a context manager that lets you enter a variable number of context managers
    - ExitStack calls the stacked context managers' `__exit__` methods in LIFO order (last entered, first exited) when the block ends
    - use when it is unknown how many context managers you need to enter in your with block

Using @contextmanager

- @contextmanager decorator reduces boilerplate code needed to create a context manager
    - just implement a generator with a single yield that should produce whatever you want the `__enter__` method to return
- yield is used to split the body of the function in two parts
    - everything before the yield will be executed at the beginning of the with block when the interpreter calls `__enter__`
    - code after yield will run when `__exit__` is called at the end of the block
- the contextmanager decorator wraps the function in a class that implements the `__enter__` and `__exit__` methods
    - `__enter__` method
        a. invokes the generator function and holds on to the generator object
        b. calls `next()` on the generator object to make it run to the yield keyword
        c. returns the value yielded by the call so it can be bound to a target variable in the with/as form

- `__exit__` method
    a. checks if an exception was passed as exc_type
        - if so `gen.throw(exception)` is invoked, causing the exception to be raised in the yield line inside the generator function body
    b. otherwise, `next(gen)` is called, resuming the execution of the generator function body after the yield
- NOTE: a try/finally (or a with block) around the yield is unavoidable when using @contextmanager
    - users of the context manager may do anything within the with block

Ch. 16 - Coroutines

- a coroutine (syntactically like a generator) is just a function with the yield keyword
    - yield usually appears on the right side of an expression (e.g., datum = yield)
    - it may or may not produce a value
        - if there is no expression after the yield keyword, the generator yields None
- may receive data from the caller, which uses `.send(datum)` instead of `next(...)` to feed the coroutine
    - caller pushes values into the coroutine
    - possible that no data goes in or out through the yield keyword
- yield is a control flow device that can be used to implement cooperative multitasking
    - each coroutine yields control to a central scheduler so that other coroutines can be activated
- to understand coroutines think of yield primarily in terms of control flow

How Coroutines Evolved from Generators

- infrastructure for coroutines appeared in PEP 342 in Python 2.5 (2006)
- yield keyword can be used in an expression
- the `.send(value)` method was added to the generator API
- caller of the generator can post data using send that then becomes the value of the yield expression inside the generator function
    - allows a generator to be used as a coroutine
- coroutine can be seen as a procedure that collaborates with the caller, yielding and receiving values from the caller
- PEP 342 also added .throw() and .close() methods that respectively allow the caller to throw an exception to be handled inside the generator and to terminate it . These features are covered in the next section and in "Coroutine Termination and Exception Handling".
- latest evolutionary step for coroutines - PEP 380 - Syntax for Delegating to a Subgenerator
    - Python 3.3 (2012)

- generator can now return a value
    - previously, providing a value to the return statement inside a generator raised a SyntaxError
- `yield from` enables complex generators to be refactored into smaller, nested generators while avoiding a lot of boilerplate code previously required for a generator to delegate to subgenerators

Basic Behavior of a Generator Used as a Coroutine

- define a function with yield in the body where yield is used in an expression
    - when the coroutine is designed just to receive data from the client it yields None
        - implicit when there is no expression to the right of the yield keyword
- call the function to get a generator object back
- the first call is `next()` because the generator hasn't started so it's not waiting in a yield and it cannot be sent any data initially
    - call to next makes the yield in the coroutine body evaluate
    - coroutine resumes and runs until the next yield or termination
- if control flows off the end of the coroutine body the generator machinery will raise StopIteration
- a coroutine can be in one of four states
- determine the current state using the `inspect.getgeneratorstate()`
    - 'GEN_CREATED' - waiting to start execution
    - 'GEN_RUNNING' - currently being executed by the interpreter
    - 'GEN_SUSPENDED' - currently suspended at a yield expression
    - 'GEN_CLOSED' - execution has completed
- the argument to the send method will become the value of the pending yield expression
    - can only make a call to send if the coroutine is currently suspended
    - not the case if the coroutine has never been activated ('GEN_CREATED')
    - the first activation of a coroutine is always done with next
        - can also call `.send(None)` and the effect is the same
- execution of the coroutine is suspended exactly at the yield keyword

Example: Coroutine to Compute a Running Average

- see book for code
- one benefit of coroutines is that state can be simple local variables
    - no instance attributes or closures

Decorators for Coroutine Priming

- a priming decorator is sometimes used to make coroutine usage more convenient (i.e. avoid the need to call next before being able to call send)
- coroutil module - coroutine decorator
- many other coroutine decorators like tornado.gen

Coroutine Termination and Exception Handling

- unhandled exception in a coroutine propagates to the caller of the next or send that triggered it
    - any attempt to reactivate will trigger StopIteration
- to terminate coroutines
    - can use send with some sentinel value that tells the coroutine to exit
        - e.g. None and Ellipsis
- generator objects have two methods that allow the client to explicitly send exceptions into the coroutine since Python 2.5
    - `generator.throw(exc_type[, exc_value[, traceback]])`
        - causes the yield expression where the generator was paused to raise the exception given
        - if the exception is handled by the generator, flow advances to the next yield, and the value yielded becomes the value of the `generator.throw` call
        - if the exception is not handled by the generator, it propagates to the context of the caller
    - `generator.close()`
        - causes the yield expression where the generator was paused to raise a GeneratorExit exception
        - no error is reported to the caller if the generator does not handle that exception or raises StopIteration — usually by running to completion
        - when receiving a GeneratorExit, the generator must not yield a value, otherwise a RuntimeError is raised
        - if any other exception is raised by the generator, it propagates to the caller

Returning a Value from a Coroutine

- can return a value from a coroutine by returning a value normally
    - before Python 3.3. it was a syntax error to return a value in a generator function
- can hack the return by passing it with StopIteration
    - defined as part of PEP 380
    - the yield from construct handles it automatically by catching StopIteration internally

    - interpreter not only consumes the StopIteration, but its value attribute becomes the

Using yield from

- similar language constructs are called `await` in other languages, which conveys the purpose much better
    - `yield from another_co()` allows the other coroutine to take over and will yield values to the outer called
- the use of yield from requires a nontrivial arrangement of code
- PEP 380 uses additional terms
    - delegating generator
        - the generator function that contains the yield from expression
    - subgenerator
        - the generator obtained from the part of the yield from expression
    - caller
        - client code that calls the delegating generator . Depending on context, I use
- client vs. caller distinguishes from the delegating generator, which is also a caller (it calls the subgenerator)
- NOTE: PEP 380 often uses the word iterator to refer to the subgenerator
    - confusing because the delegating generator is also an iterator
    - stick to subgenerator
    - subgenerator can be a simple iterator implementing only `__next__`
    - yield from can handle that as well
- see text for code and details
- if a subgenerator never terminates, the delegating generator will be suspended forever at the yield from
    - will not prevent program from making progress because the yield from transfers control to the client code
    - means that some task will be left unfinished
- every `yield from` chain must be driven by a call to next or send in the client code

The Meaning of yield from

- ok first approximation at a definition
    - when the iterator is another generator, the effect is the same as if the body of the subgenerator were inlined at the point of the yield from expression
    - the subgenerator is allowed to execute a return statement with a value
        - that value becomes the value of the yield from expression
- more details
    - any values that the subgenerator yields are passed directly to the caller of the delegating generator (i.e., the client code)

    - any values sent to the delegating generator using `send()` are passed directly to the subgenerator
        - if the sent value is None, the subgenerator's `__next__()` method is called
        - if the sent value is not None, the subgenerator's `send()` method is called
        - if the call raises StopIteration, the delegating generator is resumed
        - any other exception is propagated to the delegating generator
    - return expr in a generator (or subgenerator) causes StopIteration(expr) to be raised upon exit from the generator
    - the value of the yield from expression is the first argument to the StopIteration exception raised by the subgenerator when it terminates
    - exceptions other than GeneratorExit thrown into the delegating generator are passed to the `throw()` method of the subgenerator
        - if the call raises StopIteration, the delegating generator is resumed
        - any other exception is propagated to the delegating generator
    - if a GeneratorExit exception is thrown into the delegating generator, or the close() method of the delegating generator is called, then the close() method of the subgenerator is called if it has one
        - if this call results in an exception, it is propagated to the delegating generator
        - otherwise, GeneratorExit is raised in the delegating generator
- yield from behavior is documented using pseudocode (with Python syntax) in PEP 380
- see text for more explanation here

Use Case: Coroutines for Discrete Event Simulation

- coroutines are the fundamental building blocks of asyncio
- event simulation shows how to implement concurrent activities using coroutines instead of threads
- see book for description of discrete event simulation and code

Summary

- generators can be used to write
    - traditional "pull" style (iterators) (see ch. 14)
    - "push" style (current chapter)
    - "tasks" (event example and ch. 18)

Ch. 17 - Concurrency with Futures

- available in 3.2 but 2.5 and up as the futures package

Example: Web Downloads in Three Styles

- network IO requires concurrency
- three flavors here
  - sequential
  - threadpools and futures
  - asyncio
- NOTE: be careful not to inadvertently launch a DoS attack when testing against the public web
  - set up your own test server for non-trivial HTTP client testing
- sequential
  - no error handling for now
  - see text for code
- NOTE: the requests library is available on PyPI
  - more powerful and easier to use than the urllib.request module from the Python 3 standard library
  - considered a model Pythonic API
  - compatible with Python 2.6 and up
  - urllib2 from Python 2 was moved and renamed in Python 3
    - more convenient to use requests regardless of Python version
- concurrent.futures
  - ThreadPoolExecutor and ProcessPoolExecutor classes
    - implement an interface that allows submitting callables for execution in different threads or processes
    - classes manage an internal pool of worker threads or processes and a queue of tasks to be executed
    - interface is very high level
    - don't need to know about any of those details for a simple use case
  - see text for code
- futures are main components in concurrent.futures and of asyncio
- two classes named Future in the standard library (as of 3.4)
  - concurrent.futures.Future
  - asyncio.Future

- an instance of either Future class represents a deferred computation that may or may not have completed
  - similar
    - Deferred class in Twisted
    - Future class in Tornado
    - Promise objects in JavaScript
- encapsulate pending operations so
  - they can be put in queues
  - state of completion can be queried
  - results (or exceptions) can be retrieved when available
- meant to be instantiated exclusively by the concurrency framework
  - concurrent.futures.Future instances are created only as the result of scheduling something for execution with a concurrent.futures.Executor subclass
  - Executor.submit() method takes a callable, schedules it to run, and returns a future . Client code is not supposed to change the state of a future: the concurrency framework changes the state of a future when the computation it represents is done, and we
- both types of Future have a `.done()` method that is nonblocking and returns a Boolean signaling whether the callable linked to that future has executed or not
  - client code usually asks to be notified
  - `.add_done_callback()` method - passed a callable and the callable will be invoked with the future as the single argument when the future is done
  - `.result()` method - returns the result of the callable, or re-raises whatever exception might have been thrown when the callable was executed (same in both classes) . However, when the future is not done, the
  - behavior of the `result` method is different between the two Futures when the future is not done
    - concurrency.futures.Future - invoking f.result() will block the caller's thread until the result is ready
      - optional timeout argument can be passed, and if the future is not done in the specified time, a TimeoutError exception is raised
    - asyncio.Future.result method does not support timeout
      - preferred way to get the result of futures in that library is to use yield from
- several functions in both libraries return futures
- NOTE: expect timing variability when changing `max_workers` and working with concurrent code in general

- none of the concurrent scripts mentioned so far can perform downloads in parallel
  - concurrent.futures are limited by the GIL
  - the asyncio example is single threaded

Blocking I/O and the GIL

- GIL handles thread safety in CPython by only executing Python bytecodes on one thread at a time
  - does not take advantage of several CPU cores at once
- Python code has no control over the GIL
  - built-in functions or an extension written in C can release the GIL while running time-consuming tasks
  - Python library coded in C can manage the GIL, launch its own OS threads, and take advantage of all available CPU cores
- all standard library functions that perform blocking I/O release the GIL when waiting for a result from the OS
  - Python programs that are I/O bound can benefit from using threads at the Python level
  - blocked I/O function releases the GIL so another thread can run while one Python thread is waiting for a response from the network
- NOTE: every blocking I/O function in the Python standard library releases the GIL
  - time.sleep() function also releases
  - Python threads are still useful in I/O bound applications despite the GIL

Launching Processes with concurrent.futures

- concurrent.futures does enable truly parallel computations because it supports distributing work among multiple Python processes using the ProcessPoolExecutor class
  - bypasses the GIL and leverages all available CPU cores
  - good for CPU-bound processing
- ProcessPoolExecutor and ThreadPoolExecutor implement the generic Executor interface
  - easy to switch from a thread-based to a process-based solution using concurrent.futures
- no advantage in using a ProcessPoolExecutor for any I/O-bound job with futures.ProcessPoolExecutor() as executor: For simple uses, the
- only notable difference between the two concrete executor classes is that `ThreadPoolExecutor.__init__` requires a max_workers argument setting the number of threads in the pool
  - optional argument in ProcessPoolExecutor
    - default is the number of CPUs returned by os.cpu_count()

    - often not used . This makes sense: for CPU-bound processing, it makes no sense to ask for more workers than CPUs. On the other hand,
  - can use 10, 100, or 1,000 threads in a ThreadPoolExecutor for I/O-bound processing
    - best number depends on available memory
    - finding the optimal number will require careful testing
- NOTE: try PyPy for CPU-intensive tasks

Experimenting with Executor.map

- map(func, *iterables, timeout=None, chunksize=1)
- equivalent to map(func, *iterables)
  - func is executed asynchronously and several calls to func may be made concurrently
  - returned iterator raises a concurrent.futures.TimeoutError if `__next__()` is called and the result isn't available after timeout seconds from the original call to Executor.map ()
  - timeout can be an int or a float
  - if timeout is not specified or None, there is no limit to the wait time
  - if a call raises an exception, then that exception will be raised when its value is retrieved from the iterator
  - method chops iterables into a number of chunks which it submits to the pool as separate tasks when using ProcessPoolExecutor
- easy to use but it has a feature that may or may not be helpful
  - returns the results exactly in the same order as the calls are started
  - your code will block in order as it tries to retrieve the first result of the generator returned by map
  - will get the remaining results without blocking because they will be done
- often it's preferable to get the results as they are ready regardless of the order they were submitted
  - requires a combination of the Executor.submit method and the futures.as_completed
- NOTE: the combination of executor.submit and futures.as_completed is more flexible than executor.map
  - can submit different callables and arguments
  - executor.map is designed to run the same callable on the different arguments
  - set of futures passed to futures.as_completed may come from more than one executor

Downloads with Progress Display and Error Handling

- updated script examples
- TQDM module - instantly shows a progress bar in the console when iterating through an iterator

- see text for code and details
- futures.as_completed returns an iterator that yields futures as they are done
- an idiom that is useful with futures.as_completed
  - build a dict to map each future to other data that may be useful when the future is completed
  - makes it easy to do follow-up processing with the result of the futures despite the fact that they are produced out of order
- Python threads are well suited for I/O-intensive applications
  - concurrent.futures package makes them trivially simple to use for certain use cases
- Python has supported threads since release 0.9.8 (1993)
  - concurrent.futures is the latest way of using them
- original thread module was deprecated in favor of the higher-level threading module in Python 3
- futures.ThreadPoolExecutor is not flexible enough for a certain job
  - may need a custom solution out of basic threading components such as Thread, Lock, Semaphore, etc.
    - also thread-safe queues of the queue module for passing data between threads
- moving parts are encapsulated by futures.ThreadPoolExecutor
- sidestep the GIL by launching multiple processes using futures.ProcessPoolExecutor for CPU-bound work
- multiprocessing package emulates the threading API but delegates jobs to multiple processes
  - can replace threading with few changes
  - offers facilities to solve the biggest challenge faced by collaborating processes
    - how to pass around data

Ch. 18 - Concurrency with asyncio

- concurrency is about dealing with lots of things at once
- parallelism is about doing lots of things at once
- one is about structure, one is about execution
- concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable
- often different definitions for "concurrency" and "parallelism"
- real parallelism must have multiple cores
- in practice, most processing happens concurrently and not in parallel
- asyncio is a package that implements concurrency with coroutines driven by an event loop
  - one of the largest and most ambitious libraries ever added to Python
- asyncio is incompatible with older versions of Python uses yield from expressions extensively

- NOTE: Trollius project is a backport of asyncio to Python 2.6 and newer
    - replaces yield from with yield and callables named From and Return

Thread Versus Coroutine: A Comparison

- NOTE: asyncio uses a strict definition of coroutine
    - a coroutine suitable for use with the asyncio API
        - must use yield from and not yield in its body
        - should be driven by a caller invoking it through yield from or by passing the coroutine to one of the asyncio functions such as `asyncio.async(...)`
        - @asyncio.coroutine decorator should be applied to coroutines
- NOTE: never use `time.sleep(...)` in asyncio coroutines
    - will block the main thread and freeze the event loop
    - use yield from `asyncio.sleep(DELAY)` if a coroutine needs to spin for a time
- use of the @asyncio.coroutine decorator is not mandatory but highly recommended
    - makes the coroutines stand out among regular functions
    - helps with debugging by issuing a warning when a coroutine is garbage collected without being yielded from
        - means some operation was left unfinished and is likely a bug
    - not a priming decorator
- notable thread vs. coroutine differences
    - asyncio.Task is roughly the equivalent of a threading.Thread
        - a Task is like a green thread in libraries that implement cooperative multitasking (gevent)
    - a Task drives a coroutine, and a Thread invokes a callable
        - Task objects aren't instantiated by client code
            - obtained by passing a coroutine to `asyncio.async(...)` or `loop.create_task(...)`
    - a Task object is already scheduled to run (e.g., by asyncio.async) when it is retrieved
        - a Thread instance must be explicitly told to run by calling its start method
    - with threads, the runnable function is a plain function and is directly invoked by the thread
        - in aysncio, the runnable is a coroutine driven by yield from
    - no API to terminate a thread from the outside because a thread could be interrupted at any point (would leave system in an invalid state)
        - tasks have the Task.cancel() instance method (raises CancelledError inside the coroutine)
        - coroutine can deal with this by catching the exception in the yield where it's suspended
    - main coroutines must be executed with loop.run_until_complete in the main function

- programming with threads can be challenging to reason about because the scheduler can interrupt a thread at any time
    - must remember to hold locks to protect the critical sections of the program to avoid getting interrupted in the middle of a multistep operation (could leave data in an invalid state)
- everything is protected against interruption by default with coroutines
    - must explicitly yield to let the rest of the program run
    - have coroutines that are "synchronized" by definition (instead of holding locks to synchronize the operations of multiple threads)
        - only one of them is running at any time
    - use yield or yield from to give control back to the scheduler
- by definition, a coroutine can only be cancelled when it's suspended at a yield point
    - can perform cleanup by handling the CancelledError exception
    - possible to safely cancel a coroutine

asyncio.Future: Nonblocking by Design

- asyncio.Future and the concurrent.futures.Future classes have mostly the same interface
    - implemented differently and are not interchangeable
    - may unify asyncio.Future and concurrent.futures.Future in the future (e.g., by adding an **iter** method to concurrent.futures.Future that works with yield from)
- futures are created only as the result of scheduling something for execution
- `BaseEventLoop.create_task(...)` takes a coroutine, schedules it to run, and returns an asyncio.Task instance
    - Task is also an instance of asyncio.Future because Task is a subclass of Future designed to wrap a coroutine
- analogous to creation of concurrent.futures.Future instances by invoking `Executor.submit(...)`
- asyncio.Future class provides
    - .done()
    - .add_done_callback(...)
    - .result()
        - takes no arguments
        - can't specify a timeout
        - if called and the future is not done, it does not block waiting for the result
            - raises asyncio.InvalidStateError

- the usual way to get the result of an asyncio.Future is to yield from it
    - automatically takes care of waiting for future to finish, without blocking the event loop
    - yield from is used to give control back to the event loop
- yield from with a future is the coroutine equivalent of the functionality offered by add_done_callback
    - instead of triggering a callback, when the delayed operation is done, the event loop sets the result of the future
    - the yield from expression produces a return value inside the suspended coroutine which allows it to resume
- these following are often not needed because asyncio.Future is designed to work with yield from
    - `future.add_done_callback(...)` because processing done after the future is done can be put in the lines that follow yield from future in the coroutine
    - future.result() because the value of a yield from expression on a future is the result (e.g., result = yield from future)
- close relationship between futures and coroutines
    - can get the result of an asyncio.Future by yielding from it
    - res = yield from foo() works if foo is a coroutine function
        - returns a coroutine object when called
    - also works if foo is a plain function that returns a Future or Task instance
- one of the reasons why coroutines and futures are often interchangeable
- to execute, a coroutine must be scheduled, and then it's wrapped in an asyncio.Task
    - to obtain a task from a coroutine
        - `asyncio.async(coro_or_future, *, loop=None)`
            - unifies coroutines and futures
            - first argument can be either one
            - returned unchanged if Future or Task
            - async calls `loop.create_task(...)` on it to create a Task if it is a coroutine
            - an optional event loop may be passed as the loop=keyword argument
            - async gets the loop object by calling `asyncio.get_event_loop()` if omitted
        - `BaseEventLoop.create_task(coro)`
            - schedules the coroutine for execution and returns an asyncio.Task object . If called on a custom subclass of BaseEventLoop,

        - the object returned may be an instance of some other Task-compatible class provided by an external library (e.g., Tornado) if called on a custom subclass of BaseEventLoop

- NOTE: `BaseEventLoop.create_task(...)` is only available in Python 3.4.2 or later
    - use asyncio.async(...) or install a more recent version of asyncio from PyPI for older versions
- many asyncio functions accept coroutines and wrap them in asyncio.Task objects automatically
    - i.e. `BaseEventLoop.run_until_complete(...)`

Downloading with asyncio and aiohttp

- asyncio only supports TCP and UDP directly
- aiohttp is a 3rd party package for HTTP
- see text for code and description
- `asyncio.wait(...)` coroutine accepts an iterable of futures or coroutines
    - wait wraps each coroutine in a Task
    - all objects managed by wait become instances of Future in some way
- calling `wait(...)` returns a coroutine/generator object
- pass a coroutine to `loop.run_until_complete(...)` to drive it
    - accepts a future or a coroutine
    - wraps a coroutine into a Task
- coroutines, futures, and tasks can all be driven by yield from
    - run_until_complete does with coroutines returned by the wait call
- wait accepts two keyword-only arguments that may cause it to return even if some of the a set of futures are not complete
    - timeout
    - return_when
- must replace every function that hits the network with an asynchronous version that is invoked with yield from, so that control is given back to the event loop when working with asyncio
- trick for understanding code using yield from
    - pretend the yield from keywords are not there
    - code is as easy to read as sequential code
- using the yield from avoids blocking because the current coroutine is suspended (i.e., the delegating generator where the yield from code is), but the control flow goes back to the event loop, which can drive other coroutines
    - when a future or coroutine is done, it returns a result to the suspended coroutine, resuming it

- when using yield from with the asyncio API the following facts remain true
  - every arrangement of coroutines chained with yield from must be ultimately driven by a caller that is not a coroutine, which invokes next(...) or .send(...) on the outermost delegating generator, explicitly or implicitly (e.g., in a for loop)
  - the innermost subgenerator in the chain must be a simple generator that uses just yield or an iterable object
- specifics
  - the coroutine chains are always driven by passing the outermost delegating generator to an asyncio API call such as `loop.run_until_complete(...)`
    - code doesn't drive a coroutine chain by calling `next(...)` or `.send(...)` on it when using asyncio
      - asyncio event loop does that
  - coroutine chains always end by delegating with yield from to some asyncio coroutine function or coroutine method (e.g., yield from asyncio.sleep(...)) or coroutines from libraries that implement higher-level protocols (e.g., resp=yield from aiohttp.request(' GET', url))
    - the innermost subgenerator will be a library function that does the actual I/O, not something written by the programmer
- asynchronous code consists of coroutines that are delegating generators driven by asyncio itself and that delegate to asyncio library coroutines when using asyncio
  - creates pipelines where the asyncio event loop drives library functions that perform low-level asynchronous I/O using programmer's coroutines

Running Circles Around Blocking Calls

- blocking function - on that does disk or network I/O
  - cannot be treated the same as nonblocking functions
- are two ways to prevent blocking calls to halt the progress of the entire application
  - run each blocking operation in a separate thread
  - turn every blocking operation into a nonblocking asynchronous call
- threads work fine, but has high memory overhead for each OS thread
  - can't afford one thread per connection for thousands of connections
- callbacks are the traditional way to implement asynchronous calls with low memory overhead
  - low-level concept
  - register a function to be called when something happens.
  - every call made can be nonblocking
- callbacks depend on underlying infrastructure that uses interrupts, threads, polling, background processes, etc. to ensure that multiple concurrent requests make progress and they eventually get done
  - event loop calls callback when it gets a response
  - single main thread shared by the event loop and application code is never blocked

- When used as coroutines,
- generators provide an alternative way to do asynchronous programming when used as coroutines

Enhancing the asyncio downloader Script

- network client code should always use some throttling mechanism to avoid hitting the server with too many concurrent requests
    - overall performance of the system may degrade if the server is overloaded
- previous throttling was done by instantiating the ThreadPoolExecutor with the required max_workers argument set so that a max number of concurrent requests limits the threads that can be started in a pool
- current example uses asyncio.Semaphore
    - Semaphore is an object that holds an internal counter that is decremented whenever the .acquire() coroutine method is called on it, and incremented when the .release() coroutine method is called
    - the initial value of the counter is set when the Semaphore is instantiated

```
semaphore = asyncio.Semaphore(concur_req)
```

- calling .acquire() does not block when the counter is greater than zero
    - if the counter is zero .acquire() will block the calling coroutine until some other coroutine calls .release() on the same Semaphore
    - increments the counter
- can use the semaphore as a context manager
    - guarantees that no more than the max number of concurrent requests instances of the coroutine will be started at any time

```
with (yield from semaphore):
    image = yield from get_flag(base_url, cc)
```

- use yield from to retrieve the results of the futures yielded by asyncio.as_completed
    - as_completed must be invoked in a coroutine
    - must move most of the previous download functionality to the coroutine
- couldn't simply turn download_many into a coroutine, because I
- may need to take care when splitting functionality between plain functions and coroutines
    - need to run the as_completed loop, the event loop and schedule a coroutine by passing it to loop.run_until_complete
- the futures returned by asyncio.as_completed are not necessarily the same futures passed into the as_completed call
    - the asyncio machinery replaces the future objects provided with others that will produce the same results

Using an Executor to Avoid Blocking the Event Loop

- Python programs often don't pay enough attention to the fact that local filesystem access is blocking
  - network latency is main consideration
  - Node.js functions have reminders that filesystem functions are blocking because their signatures require a callback
- blocking for disk I/O wastes millions of CPU cycles
- for blocking functions that run in one one of several worker threads
  - blocking I/O call releases the GIL behind the scenes so another thread can proceed
- whole application freezes while the file is being saved if block code runs on the same thread as the asyncio event loop
  - solution to this problem is the run_in_executor method of the event loop object
  - asyncio event loop has a thread pool executor behind the scenes
    - can send callables to be executed by it with run_in_executor
- NOTE: no noticeable change in performance for using run_in_executor for saving 13KB files
  - will see an effect when saving 130KB files
  - advantage of using run_in_executor becomes clear with this size
    - speedup will be significant
- advantage of coroutines over callbacks becomes evident when coordinating asynchronous requests and not just making completely independent requests

From Callbacks to Futures and Coroutines

- event-oriented programming with coroutines requires effort
  - important to be clear on how it improves on the classic callback style
- callback-style event-oriented programming often leads to "callback hell"
  - nesting of callbacks when one operation depends on the result of the previous operation
  - three asynchronous calls that happen in succession require callbacks nested three levels deep
- each function does part of the job, sets up the next callback, and returns, to let the event loop proceed
  - all local context is lost
  - don't have the value of any more when the next callback is executed
  - must rely on closures or external data structures to store context between the different stages of the processing
- within a coroutine, to perform
- three asynchronous actions in succession in a coroutine yield three times to let the event loop continue running
  - coroutine is activated with a .send() call when the result is ready
  - similar to invoking a callback from the perspective of the event loop

- situation is vastly improved with a coroutine-style asynchronous API
    - entire sequence of three operations is in one function body
    - like sequential code with local variables to retain the context of the overall task
- coroutines and yield from enable asynchronous programming without callbacks
- also provides a context for error reporting
- exception cannot be caught in an asynchronous call that returns immediately before any I/O is performed In callback-based APIs, this is
- solved by registering two callbacks for each asynchronous call in callback-based APIs
    - one for handling the result of successful operations
    - another for handling errors
- conditions in callback hell quickly deteriorate when error handling is involved
- can place all async calls for an operation in a single function
    - can place any potentially throwing calls by putting respective yield from lines inside try/except blocks
- downsides
    - must use coroutines and get used to yield from
    - can't simply call the coroutines
        - must explicitly schedule the execution of the coroutine with the event loop or activate it using yield from in another coroutine that is scheduled for execution
        - nothing would happen without the call to loop.create_task

Doing Multiple Requests for Each Download

- multiple requests in the same task is easy with threads
    - just make one request then the other, blocking the thread twice, and keeping data in local variables , ready to use when saving the files. If you need
- to do the same in an asynchronous script with callbacks
    - data needs to be passed around in a closure or held somewhere until use because each callback runs in a different local context
- coroutines and yield from provide a cleaner solution
    - not as simple as with threads
    - more manageable than chained or nested callbacks
- challenge is to know when to use yield from and when to not use it
    - yield from coroutines and asyncio.Future instances — including tasks
    - some APIs are tricky, mixing coroutines and plain functions in seemingly arbitrary ways
- experiment with example 18-13

Writing asyncio Servers

- see text for code and description of servers

- example looks like a view function in Django or Flask
  - nothing asynchronous about implementation
    - gets a request
    - fetches data from a database
    - builds a response by rendering a full HTML page
  - database is example is in memory
    - accessing a real database should be done asynchronously . For example, the
- aiopg package provides an asynchronous PostgreSQL driver compatible with asyncio
- concurrent systems must split large chunks of work into smaller pieces to stay responsive
  - in addition to avoiding blocking calls
- most queries return much smaller responses than given in example
- to avoid the long response implement pagination
  - return results with at most ~200 rows
  - user clicks or scrolls the page to fetch more
  - use AJAX or even WebSockets to send the next batch
  - most of the necessary coding for sending results in batches would be on the browser
  - smart asynchronous clients make better use of server resources
- smart clients can help even old-style Django applications
  - need frameworks that support asynchronous programming all the way
  - handling of HTTP requests, responses, and database access
  - especially true for real-time services such as games and media streaming with WebSockets

Ch. 19 - Dynamic Attributes and Properties

- importance of properties
  - existence makes it safe and advisable for to expose public data attributes as part of a class's public interface
- data attributes and methods are collectively known as attributes in Python
  - a method is just an attribute that is callable
- can also create properties, which can be used to replace a public data attribute with accessor methods (i.e., getter/ setter), without changing the class interface
  - agrees with the uniform access principle
    - all services offered by a module should be available through a uniform notation which does not betray whether they are implemented through storage or through computation
- Python provides a rich API for controlling attribute access and implementing dynamic attributes
  - interpreter calls special methods such as `__getattr__` and `__setattr__` to evaluate attribute access using dot notation (e.g., obj.attr)

    - a user-defined class implementing `__getattr__` can implement "virtual attributes" by computing values on the fly whenever somebody tries to read a nonexistent attribute

Data Wrangling with Dynamic Attributes

- see text for code and description
- example JSON class uses `__getattr__` and `hasattr` with `self.__data`
- needs special handling for attribute names that are Python keywords
    - use a mechanism that appends _ to Python keywords used as attributes
    - must also use if the string is not a valid Python identifier
        - use str.isidentifier()
- `__init__` is often referred to as the constructor method
    - the special method that actually constructs an instance is `__new__`
        - a class method (but gets special treatment, so the @classmethod decorator is not used)
        - must return an instance
        - returned instance will in turn be passed as the first argument self of `__init__`
- `__init__` is forbidden from returning anything
    - really an "initializer"
- real constructor is `__new__`
    - rarely need to code because the implementation inherited from object suffices
- `__new__` method can also return an instance of a different class
    - when that happens the interpreter does not call `__init__` . In other words, the
- the process of building an object in Python can be summarized

```
def object_maker(the_class, some_arg):
    new_object = the_class.__new__( some_arg)
    if isinstance(new_object, the_class):
        the_class.__init__(new_object, some_arg)
    return new_object

 # roughly equivalent
 x = Foo('bar')
 x = object_maker(Foo, 'bar')
```

- shelve module provides pickle sotrage

- shelve.open high-level function returns a shelve.Shelf instance
    - simple key-value object database backed by the dbm module
    - shelve.Shelf subclasses abc.MutableMapping
    - shelve.Shelf provides a few other I/O management methods, like sync and close
    - also a context manager
    - keys and values are saved whenever a new value is assigned to a key
    - keys must be strings
    - values must be objects that the pickle module can handle
- shelve provides a simple, efficient way to reorganize JSON data
- NOTE: see argparse.Namespace and multiprocessing.Namespace for classes that build arbitrary sets of attributes from keyword arguments passed to the constructor
- NOTE: In Python 2, only "new style" classes support properties
    - must subclass directly or indirectly from object
- NOTE: always the risk of bugs due to shadowing of class attributes (such as methods) or data loss through accidental overwriting of existing instance attributes when creating instance attribute names from data
    - main reason why, by default, Python dicts are not like JavaScript objects

Using a Property for Attribute Validation

- getter and setter

```
@property
def weight(self):
    return self.__weight

@weight.setter
def weight(self, value):
    if value > 0:
        self.__weight = value
    else:
        raise ValueError('value must be > 0')
```

- property setter guards against improper setter values
- methods that implement a property all have the name of the public attribute
    - actual value is stored in a private attribute
- decorated getter has a .setter attribute, which is also a decorator
    - ties the getter and setter together
- NOTE: cure for repetition is abstraction!!!
- two ways to abstract away property definitions
    - property factory
    - descriptor class
        - more flexible (ch. 20)

- properties are implemented as descriptor classes themselves

A Proper Look at Properties

- the property built-in is a class
    - functions and classes are often interchangeable in Python
    - both are callable and there is no new operator for object instantiation
    - invoking a constructor is no different than invoking a factory function
- both classes and function can be used as decorators
    - need to return a new callable that is a suitable replacement of the decorated function
- full signature of the property constructor

```
property(fget = None, fset = None, fdel = None, doc = None)
```

- all arguments are optional
- corresponding operation is not allowed by the resulting property object if a function is not provided for one of them
- property type was added in Python 2.2
    - @ decorator syntax appeared only in Python 2.4
        - properties were defined by passing the accessor functions as the first two arguments
- classic syntax for defining properties without decorators

```
# within class definition after getters and setters
weight = property(get_weight, set_weight)
```

- classic form is better than the decorator syntax in some situations

    - property factoryies, for instance

- decorators make it explicit which are the getters and setters without depending on the convention of using get and set prefixes in their names

- properties override instance attributes

- properties are always class attributes

    - manage attribute access in the instances of the class

- when an instance and its class both have a data attribute by the same name the instance attribute overrides (shadows) the class attribute (when read through that instance)

- expression like obj.attr does not search for attr starting with obj

    - starts at `obj.__class__`

    - Python looks in the obj instance itself only if there is no property named attr in the class

- applies not only to properties but
- applies to a whole category of descriptors
    - overriding descriptors
    - not only properties
- every Python code unit can have a docstring
    - modules
    - functions
    - classes
    - methods
- property documentation
- `__doc__` attribute of the property
- used by tools such as the console help() function or IDEs need to display the documentation
- with classic call syntax property can get the documentation string as the doc argument
- as a decorator, the docstring of the getter method — the one with the @property decorator itself — is used as the documentation of the property

Coding a Property Factory

- creating and using a property factory requires repetition in the class definition

```
class ClassName:
    property_name = property_factory('property_name')
```

- NOTE: improving this so user doesn't have to retype the attribute name is a nontrivial metaprogramming problem
- factory

```
def property_factory(storage_name):
    def factory_getter(instance):
        return instance.__dict__[storage_name]

    def factory_setter(instance, value):
        if condition:
            instance.__dict__[storage_name] = value
        else:
            raise Exception
    return property(factory_getter, factory_setter)
```

- the name of the attribute to store a value is hardcoded in the getter and setter methods when writing a property in the traditional way
- property factory makes getter and setter functions generic
    - depend on the storage_name variable to know where to get/set the managed attribute in the instance `__dict__`
- storage_name must be set to a unique value each time the quantity factory is called to build a property
    - the getter and setter functions will be wrapped by the property object created in the last line of the factory function
    - when called, the functions will read the storage_name from their closures to determine where to retrieve/store the managed attribute values
    - properties built by the factory leverage the behavior by how it overrides
        - property overrides the instance attribute so that every reference to self. or is handled by the property functions
        - only way to bypass the property logic is to access the instance `__dict__` directly
- same validation may appear in many fields across classes
    - factory would be placed in a utility module to be used over and over again
- simple factory could be refactored into a more extensible descriptor class
    - specialized subclasses performing different validations

Handling Attribute Deletion

- object attributes can be deleted with `del`

```
del obj.attr
```

- not common, but can delete properties with @<property_name>.deleter for the deleter function
- use `fdel` arg with the classic call syntax
- attribute deletion can also be handled by implementing the lower-level `__delattr__` special method when not using a property

Essential Attributes and Functions for Attribute Handling

- mentioned some of the built-in functions and special methods Python provides for dealing with dynamic attributes

Special Attributes that Affect Attribute Handling

The behavior of many of the functions and

- many of the functions and special methods depend on three special attributes
  - `__class__`
    - a reference to the object's class
    - `obj.__class__` is the same as type(obj) Python looks for special methods such as **getattr** only in an object's class, and not in the instances themselves.
  - `__dict__`
    - a mapping that stores the writable attributes of an object or class
    - an object that has a `__dict__` can have arbitrary new attributes set at any time
    - instances may not have a `__dict__` if a class has the `__slots__` attribute
  - `__slots__`
    - an attribute that may be defined in a class to limit the attributes its instances can have
    - a tuple of strings naming the allowed attributes
    - instances of that class will not have a `__dict__` of their own if `__dict__` is not in slots
      - only named attributes will be allowed in them

Built-In Functions for Attribute Handling

- built-in functions that perform object attribute reading, writing, and introspection
  - `dir([object])`
    - lists most attributes of the object
    - official docs say dir is intended for interactive use
      - does not provide a comprehensive list of attributes
    - can inspect objects implemented with or without a `__dict__`
    - `__dict__` attribute is not listed by dir
    - `__dict__` keys are listed
    - many special attributes of classes are not listed by dir either
      - `__mro__`
      - `__bases__`
      - `__name__`
    - dir lists the names in the current scope if the optional object argument is not given
  - `getattr(object, name[, default])`
    - gets the attribute identified by the name string from the object
    - may fetch an attribute from the object's class or from a superclass
    - raises AttributeError or returns the default value (if given) if no such attribute exists

- `hasattr(object, name)`
    - returns True if the named attribute exists in the object or can be somehow fetched through it (e.g., by inheritance)
    - implemented by calling `getattr(object, name)` and seeing whether it raises an AttributeError or not
- `setattr(object, name, value)`
    - assigns the value to the named attribute of object, if the object allows it
    - may create a new attribute or overwrite an existing one
- `vars([object])`
    - returns the **dict** of object
    - can't deal with instances of classes that define `__slots__` and don't have a `__dict__`
    - same as locals() without vars - returns a dict representing the local scope

Special Methods for Attribute Handling

- these special methods handle attribute retrieval, setting, deletion, and listing when implemented in a user-defined class
- attribute access using either dot notation or the built-in functions getattr, hasattr, and setattr trigger these special methods
- reading and writing attributes directly in the instance `__dict__` does not trigger these special methods
    - usual way to bypass them if needed
- implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary for custom classes
    - assume that the special methods will be retrieved on the class itself even when the target of the action is an instance
- special methods are not shadowed by instance attributes with the same name for this reason
    - `__delattr__( self, name)`
        - always called when there is an attempt to delete an attribute using the del statement
        - triggers `Class.__delattr__(obj, 'attr')`
    - `__dir__(self)`
        - called when dir is invoked on the object, to provide a listing of attributes
    - `__getattr__(self, name)`
        - called only when an attempt to retrieve the named attribute fails, after the obj, Class, and its superclasses are searched

    - expressions `obj.no_such_attr` , `getattr(obj, 'no_such_attr')` , and `hasattr(obj, 'no_such_attr')` may trigger `Class.__getattr__(obj, 'no_such_attr')`
        - only if an attribute by that name cannot be found in obj or in Class and its superclasses
  - `__getattribute__(self, name)`
    - always called when there is an attempt to retrieve the named attribute, except when the attribute sought is a special attribute or method
    - dot notation and the getattr and hasattr built-ins trigger this method
    - `__getattr__` is only invoked after `__getattribute__` , and only when `__getattribute__` raises AttributeError
    - to retrieve attributes of the instance obj without triggering an infinite recursion, implementations of `__getattribute__` should use `super().__getattribute__(obj, name)`
  - `__setattr__(self, name, value)`
    - always called when there is an attempt to set the named attribute
    - dot notation and the setattr built-in trigger this method
    - `Class.__setattr__(obj, 'attr', 42)`
- NOTE: `__getattribute__` and `__setattr__` special methods are harder to use correctly than `__getattr__` (only handles nonexisting attribute names) because they are unconditionally called and affect practically every attribute access
  - properties or descriptors is less error prone than defining these special methods

Ch. 20 - Attribute Descriptors

- good for gaining a deeper understanding of how Python works
- good for reuse of access logic across multiple attributes
- a descriptor is a class that implements a protocol consisting of the `__get__` , `__set__` , and `__delete__` methods
  - property class implements the full descriptor protocol
  - partial implementations are OK
  - most descriptors implement only `__get__` and `__set__`
  - many implement only one of these methods
- other Python features that leverage descriptors are methods and the classmethod and staticmethod decorators
- key to Python mastery

Descriptor Example: Attribute Validation

- property factory - function to avoid duplication for property creation

- OO method - descriptor class
- a descriptor is a class implementing
- descriptor class
  - a class implementing the descriptor protocol
  - a `__get__` , a `__set__` , or a `__delete__` special method
- managed class
  - the class where the descriptor instances are declared as class attributes
- descriptor instance
  - each instance of a descriptor class, declared as a class attribute of the managed class
- managed instance
  - one instance of the managed class
- storage attribute
  - an attribute of the managed instance that will hold the value of a managed attribute for that particular instance
  - distinct from the descriptor instances, which are always class attributes
- managed attribute
  - a public attribute in the managed class that will be handled by a descriptor instance, with values stored in storage attributes
  - a descriptor instance and a storage attribute provide the infrastructure for a managed attribute
- NOTE: when coding a **set** method keep in mind
  - self is the descriptor instance
  - instance is the managed instance
  - descriptors managing instance attributes should store values in the managed instances
- do not store the values on self instead of instance
  - anything stored in the descriptor is actually part of a class attribute rather than an instance
- can generate unique storage names for each descriptor to avoid retyping names

```
class Descriptor:
    __counter = 0
    def __init__(self):
        cls = self.__class__
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = '_{0}#{1}'.format(prefix, index)
        cls.__counter += 1
```

- use the higher-level getattr and setattr built-ins to store the value instead of resorting to `instance.__dict__`
  - the managed attribute and the storage attribute have different names
  - calling getattr on the storage attribute will not trigger the descriptor (avoids the recursion)
- NOTE: following the convention Python uses to do name mangling (e.g., _LineItem__quantity0) requires the name of the managed class
  - the body of a class definition runs before the class itself is built by the interpreter
    - information is not available when each descriptor instance is created
  - may not be a need to include the managed class name to avoid accidental overwriting in subclasses
    - the descriptor class __counter will be incremented every time a new descriptor is instantiated
    - gaurantess that each storage name will be unique across all managed classes
- `__get__` receives three arguments
  - self
  - instance
  - owner
    - a reference to the managed class
    - useful when the descriptor is used to get attributes from the class
- if a managed attribute is retrieved via the class like LineItem.weight, the descriptor `__get__` method receives None as the value for the instance argument
- good practice to make `__get__` return the descriptor instance when the managed attribute is accessed through the class to support introspection and other metaprogramming tricks by the user
- prefer the descriptor class for two reasons
  - a descriptor class can be extended by subclassing
    - reusing code from a factory function without copying and pasting is much harder
  - more straightforward to hold state in class and instance attributes than in function attributes and closures
- property factory code does not depend on strange object relationships evidenced by descriptor methods having arguments named self and instance
- property factory pattern is simpler but the descriptor class approach is more extensible and more widely used

- example 5 is an example of the template method
    - defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior

Overriding Versus Nonoverriding Descriptors

- important asymmetry Python attributes handling
    - reading an attribute through an instance normally returns the attribute defined in the instance
        - a class attribute will be retrieved if there is no such attribute in the instance
    - assigning to an attribute in an instance normally creates the attribute in the instance without affecting the class at all
- asymmetry also affects descriptors
    - creates two broad categories of descriptors depending on whether the `__set__` method is defined
- overriding descriptor
    - a descriptor that implements the `__set__` method
    - a descriptor implementing `__set__` will override attempts to assign to instance attributes even though it is a class attribute
    - properties are also overriding descriptors
        - default `__set__` from the property class will raise AttributeError to signal that the attribute is read-only when a setter function isn't written to override it
- overriding descriptor without `__get__`
    - overriding descriptors usually implement both `__set__` and `__get__`
    - also possible to implement only `__set__`
    - only writing is handled by the descriptor
    - reading the descriptor through an instance will return the descriptor object itself because there is no `__get__` to handle access
    - the `__set__` method will still override further attempts to set that attribute if a namesake instance attribute is created with a new value via direct access to the instance `__dict__`
        - reading that attribute will simply return the new value from the instance instead of returning the descriptor object
    - e.g. the instance attribute will shadow the descriptor, but only when reading
- nonoverriding descriptor
    - a descriptor that does not implement **set**
    - setting an instance attribute with the same name will shadow the descriptor
        - renders it ineffective for handling that attribute in that specific instance
    - methods are implemented as nonoverriding descriptors
- NOTE: Python contributors and authors use different terms when discussing these concepts
    - overriding descriptors are also called data descriptors or enforced descriptors

    - nonoverriding descriptors are also known as nondata descriptors or shadowable descriptors
- the setting of attributes in the class cannot be controlled by descriptors attached to the same class
    - means that the descriptor attributes themselves can be clobbered by assigning to the class
- overwriting a descriptor in the class
    - descriptor can be overwritten by assignment to the class regardless of whether a descriptor is overriding or not
    - monkey-patching technique
    - any descriptor can be overwritten on the class itself
- another asymmetry regarding reading and writing attributes
    - reading of a class attribute can be controlled by a descriptor with `__get__` attached to the managed class
    - the writing of a class attribute cannot be handled by a descriptor with `__set__` attached to the same class
- NOTE: attach descriptors to the class of the class (metaclass) to control the setting of attributes in a class
    - the default metaclass of user-defined classes is type
        - cannot add attributes to type

Methods Are Descriptors

- a function within a class becomes a bound method because all user-defined functions have a `__get__` method
    - operate as descriptors when attached to a class
- instance and class retrieve different objects
    - the `__get__` of a function returns a reference to itself when the access happens through the managed class (as usual with descriptors)
    - the `__get__` of the function returns a bound method object when the access goes through an instance
        - a callable that wraps the function and binds the managed instance (e.g., obj) to the first argument of the function (i.e., self)
        - like functools.partial
- the bound method object also has a `__call__` method
    - handles the actual invocation
    - calls the original function referenced in `__func__` , passing the `__self__` attribute of the method as the first argument
    - how the implicit binding of the conventional self argument works
- the way functions are turned into bound methods is a prime example of how descriptors are used as infrastructure in the language

Descriptor Usage Tips

- use property to keep it simple
    - property built-in actually creates overriding descriptors implementing both `__set__` and `__get__`
        - even if a setter method is not defined
    - the default `__set__` of a property raises AttributeError
    - a property is the easiest way to create a read-only attribute
    - avoids issues below
- read-only descriptors require `__set__`
    - remember to code both `__get__` and `__set__` when using a descriptor class to implement a read-only attribute
        - setting a namesake attribute on an instance will shadow the descriptor
    - the `__set__` method of a read-only attribute should just raise AttributeError with a suitable message
- validation descriptors can work with `__set__` only
    - the `__set__` method should check the value argument it gets in a descriptor designed only for validation
        - if valid, set it directly in the instance `__dict__` using the descriptor instance name as key
    - reading the attribute with the same name from the instance will be as fast as possible
        - will not require a `__get_`
- caching can be done efficiently with `__get__` only
    - creates a nonoverriding descriptor if just `__get__` is coded
    - useful for making expensive computations and then cache the results by setting an attribute by the same name on the instance
    - the namesake instance attribute will shadow the descriptor
        - subsequent access to that attribute will fetch it directly from the instance `__dict__` and not trigger the descriptor `__get__` anymore
- nonspecial methods can be shadowed by instance attributes
    - functions and methods do not handle attempts at setting instance attributes with the same name because they only only implement `__get__`
        - simple assignment like means that further access to the method through the instance will retrieve the overriding value without affecting the class or other instances
    - this issue does not interfere with special methods
        - the interpreter only looks for special methods in the class itself
        - `repr(x)` is executed as `x.__class__.__repr__(x)`
        - a `__repr__` attribute defined in x has no effect on `repr(x)`

    - the existence of an attribute named `__getattr__` in an instance will not subvert the usual attribute access algorithm
- when doing a lot of dynamic attribute creation, where the attribute names come from data not under control of the programmer, be aware of the ability to override special methods and implement some filtering or escaping of the dynamic attribute names to preserve security . Note
- NOTE:
    - class methods are safe as long as they are always accessed through the class
    - use of only special methods, class methods, static methods, and properties is also safe
    - properties are data descriptors, so cannot be overridden by instance attributes

Descriptor docstring and Overriding Deletion

- docstring of a descriptor class is used to document every instance of the descriptor in the managed class
    - may be unsatisfactory
- can handle attempts to delete a managed attribute by implementing a `__delete__` alongside or instead of the usual `__get__` and/or `__set__` in the descriptor class

Ch. 21 - Class Metaprogramming

- metaclasses are deeper than most users should ever worry about
- class metaprogramming is the art of creating or customizing classes at runtime
    - function can be used to create a new class at any time without using the class keyword
    - class decorators are also functions that are capable of inspecting, changing, and even replacing the decorated class with another class
- metaclasses are the most advanced tool for class metaprogramming
    - allow creation of whole new categories of classes with special traits
        - abstract base classes
    - powerful, but hard to get right
- class decorators solve many of the same problems more simply
- metaclasses are now so hard to justify in real code that my favorite
- crucial distinction between import time and runtime

A Class Factory

- collections.namedtuple
    - function that, given a class name and attribute names creates a subclass of tuple that allows retrieving items by name and provides a nice `__repr__` for debugging
- see code for example of a class factory

- invoking type with three arguments is a common way of creating a class dynamically
- source code for collections.namedtuple
  - there is _class_template, a source code template as a string, and the namedtuple function fills its blanks calling `_class_template.format(...)`
  - resulting source code string is then evaluated with the exec built-in function
- NOTE: good practice to avoid exec or eval for metaprogramming in Python
  - pose serious security risks if they are fed strings (even fragments) from untrusted sources
  - Python offers sufficient introspection tools to make exec and eval unnecessary most of the time
  - Python core developers chose to use exec when implementing namedtuple
    - makes the code generated for the class available in the ._source attribute
- instances of classes created by the example have a limitation
  - not serializable
  - can't be used with the dump/load functions from the

A Class Decorator for Customizing Descriptors

When we left the LineItem example in "LineItem Take #5: A New Descriptor Type", the

- issue of descriptive storage names for descriptors
  - the value of attributes such as weight was stored in an instance attribute named _Quantity#0, which made debugging hard
  - can retrieve the storage name from a descriptor
  - would be better if the storage names actually included the name of the managed attribute
  - could not use descriptive storage names because when the descriptor is instantiated it has no way of knowing the name of the managed attribute (i.e., the class attribute to which the descriptor will be bound)
  - can inspect the class and set proper storage names to the descriptors once the whole class is assembled and the descriptors are bound to the class attributes
  - could be done in the **new** method of the class
    - correct storage names are set by the time the descriptors are used in the **init** metho d
  - using **new** for that purpose causes the logic of **new** to run every time a new instance is created, but the binding of the descriptor to the managed attribute will never change once the class itself is built
  - need to set the storage names when the class is created
- can be done with a class decorator or a metaclass
- class decorator is very similar to a function decorator
  - a function that gets a class object and returns the same class or a modified one

  - are a simpler way of doing something that previously required a metaclass
    - customizes a class the moment it's created
  - a drawback of class decorators is that they act only on the class where they are directly applied
  - means subclasses of the decorated class may or may not inherit the changes made by the decorator, depending on what those changes are

What Happens When: Import Time Versus Runtime

- must be aware of when the Python interpreter evaluates each block of code
- at import time
  - the interpreter parses the source code of a .py module in one pass from top to bottom
  - generates the bytecode to be executed
  - when syntax errors may occur
  - steps are skipped because the bytecode is ready to run if there is an up-to-date .pyc file available in the local `__pycache__` . Although
- compiling is definitely an import-time activity
- other things may also happen at import-time
  - almost every statement in Python is executable
    - they potentially run user code and change the state of the user program
  - the import statement is not just a declaration
    - runs all the top-level code of the imported module when it's imported for the first time in the process
    - any more imports of the same module will use a cache and only name binding occurs then
  - top-level code may do anything, including typical "runtime" actions
- import statement can trigger all sorts of "runtime" behavior
- top-level code
  - interpreter executes a def statement on the top level of a module when the module is imported
    - interpreter compiles the function body (on first import)
    - binds the function object to its global name
    - it does not execute the body of the function
- means that the interpreter defines top-level functions at import time, but executes their bodies only when — and if — the functions are invoked at runtime
- for classes
  - at import time, the interpreter executes the body of every class, even the body of classes nested in other classes
  - execution of a class body means that the attributes and methods of the class are defined, and then the class object itself is built
- body of classes is "top-level code" - it runs at import time

The Evaluation Time Exercises

- review these exercises
- the effects of a class decorator may not affect subclasses
- metaclasses are more effective to customize a whole class hierarchy, and not one class at a time

Metaclasses 101

- a metaclass is a class factory written as a class
- classes are objects
  - each class must be an instance of some other class
- Python classes are instances of type by default
  - type is the metaclass for most built-in and user-defined classes
  - type is an instance of itself to avoid infinite recursion
- classes do not inherit from type
  - class objects are themselves instances of type
  - all are subclasses of object
- NOTE: the classes object and type have a unique relationship
  - object is an instance of type
  - type is a subclass of object
  - relationship cannot be expressed in Python because either class would have to exist before the other could be defined
- fact that type is an instance of itself is also magical
- a few other metaclasses exist in the standard library, such as ABCMeta and Enum
- class Iterable is abstract, but ABCMeta is not — after all, Iterable is an instance of ABCMeta
  - ABCMeta is also type
- every class is an instance of type, directly or indirectly, but only metaclasses are also subclasses of type
- most important relationship to understand metaclasses
  - a metaclass, such as ABCMeta, inherits from type the power to construct classes
- important takeaway
  - all classes are instances of type
  - metaclasses are also subclasses of type
    - act as class factories
- a metaclass can customize its instances by implementing `__init__`
  - `__init__` method can do everything a class decorator can do
  - effects are more profound

- NOTE: further class customization can be done by implementing `__new__` in a metaclass
  - implementing **init** is usually enough

A Metaclass for Customizing Descriptors

- see model.Entity

The Metaclass prepare Special Method

- sometimes necessary to know the order in which the attributes of a class are defined
- both the type constructor and the `__new__` and `__init__` methods of metaclasses receive the body of the class evaluated as a mapping of names to attributes . However, by default, that
  - mapping is a dict by default
  - means the order of the attributes as they appear in the class body is lost by the time our metaclass or class decorator can look at them
- the solution to this problem is the `__prepare__` special method
  - introduced in Python 3
  - relevant only in metaclasses
  - must be a class method (i.e., defined with the @classmethod decorator)
  - invoked by the interpreter before the `__new__` method in the metaclass to create the mapping that will be filled with the attributes from the class body
  - gets the metaclass as first argument, the name of the class to be constructed and its tuple of base classes
  - must return a mapping, which will be received
- metaclasses are used in frameworks and libraries that help programmers perform
  - attribute validation
  - applying decorators to many methods at once
  - object serialization or data conversion
  - object-relational mapping
  - object-based persistency
  - dynamic translation of class structures from other languages

Classes as Objects

- every class has a number of attributes defined in the Python data model
  - see "Special Attributes" of the "Built-in Types" chapter in the Library Reference
- already looked at `__mro__` , `__class__` , and `__name__`
- `cls.__bases__`
  - tuple of base classes of the class

- `cls.__qualname__`
  - new attribute in Python 3.3 holding the qualified name of a class or function
    - dotted path from the global scope of the module to the class definition
  - specification for this attribute is PEP-3155
- `cls.__subclasses__()`
  - method returns a list of the immediate subclasses of the class
  - implementation uses weak references to avoid circular references between the superclass and its subclasses — which hold a strong reference to the superclasses in their `__bases__` attribute
  - returns the list of subclasses that currently exist in memory
- `cls.mro()`
  - interpreter calls this method when building a class to obtain the tuple of superclasses that is stored in the `__mro__` attribute of the class
  - a metaclass can override this method to customize the method resolution order of the class under construction
- NOTE: none of the attributes mentioned in this section are listed by the `dir(...)` function