

PYTHON 3: DEEP DIVE

PART 2



What this course is about

the Python language

→ canonical CPython 3.6+ implementation

the standard library

becoming an expert Python developer

idiomatic Python

obtaining a deeper understanding of the Python language

and the standard library

this is NOT an introductory course

→ refer to prerequisites video or course description



Included Course Materials

lecture videos

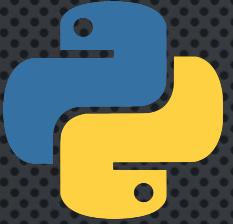
coding videos

Jupyter notebooks

projects and solutions

github repository for all code

<https://github.com/fbaptiste/python-deepdive>



Sequence Types

what are sequences?

slicing → ranges

shallow vs deep copy

the sequence protocol

implementing our own sequence types

list comprehensions → closures

sorting → sort key functions



Iterables and Iterators

more general than sequence types

differences between iterables and iterators

lazy vs eager iterables

the iterable protocol

the iterator protocol

writing our own custom iterables and iterators



Generators

what are generator?

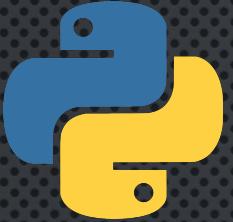
generator functions

generator expressions

the `yield` statement

the `yield from` statement

how generators are related to iterators



Iteration Tools

Many useful tools for functional approach to iteration

- built-in
- `itertools` module
- `functools` module

Aggregators

Slicing iterables

Selection and filtering

Infinite iterators

Mapping and reducing

Grouping

Combinatorics



Context Managers

what are context managers?

the context manager protocol

why are they so useful?

creating custom context managers using the context manager protocol

creating custom context managers using generator functions



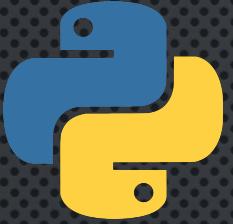
Projects

project after each section

should attempt these yourself first – practice makes perfect!

solution videos and notebooks provided

- my approach
- more than one approach possible



Extras

will keep growing over time

important new features of Python 3.6 and later

best practices

random collection of interesting stuff

additional resources

send me your suggestions!

PREREQUISITES

Copyright © 2014

Python 3: Deep Dive (Part 2) - Prerequisites

This course assumes that you have **in-depth** knowledge of the following:

functions and function arguments

```
def my_func(p1, p2, *args, k1=None, **kwargs)
```

lambdas

```
lambda x, y: x+y
```

packing and unpacking iterables

```
my_func(*my_list)
```

```
f, *_ , l = (1, 2, 3, 4, 5)
```

closures

nested scopes free variables

decorators

```
@my_decorator      @my_decorator(p1, p2)
```

Boolean truth values

```
bool(obj)
```

named tuples

```
namedtuple('Data', 'field_1 field_2')
```

`==` vs `is`

```
id(obj)
```

Python 3: Deep Dive (Part 2) - Prerequisites

This course assumes that you have **in-depth** knowledge of the following:

zip `zip(list1, list2, list3)`

map `map(lambda x: x**2, my_list)`

reduce `reduce(lambda x, y: x * y, my_list, 10)`

filter `filter(lambda p: p.age > 18, persons)`

sorted `sorted(persons, lambda p: p.name.lower())`

imports `import math`
`from math import sqrt, sin`
`from math import sqrt as sq`
`from math import *`

Python 3: Deep Dive (Part 2) - Prerequisites

You should have a **basic** understanding of creating and using classes in Python

```
class Person:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    @property  
    def age(self):  
        return self._age  
  
    @age.setter  
    def age(self, age):  
        if value <= 0:  
            raise ValueError('Age must be greater than 0')  
        else:  
            self._age = age
```

Python 3: Deep Dive (Part 2) - Prerequisites

You should understand how special functionality is implemented in Python using special methods

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __repr__(self):  
        return f'Point(x={self.x}, y={self.y})'  
  
    def __eq__(self, other):  
        if not isinstance(other, Point):  
            return False  
        else:  
            return self.x == other.x and self.y == other.y  
  
    def __gt__(self, other):  
        if not isinstance(other, Point):  
            return NotImplemented  
        else:  
            return self.x ** 2 + self.y ** 2 > other.x**2 + other.y**2  
  
    def __add__(self, other):  
        ...
```

Python 3: Deep Dive (Part 2) - Prerequisites

You should also have a basic understanding of:

for loops, while loops

break continue else

branching

if ... elif... else...

exception handling

```
try:  
    my_func()  
except ValueError as ex:  
    handle_value_error()  
finally:  
    cleanup()
```

PYTHON TOOLS NEEDED

This course is all about the Python language and the standard library

I do not make use of any 3rd party library

EXCEPT

Jupyter Notebooks

- I can provide you fully annotated code
- all notebooks are downloadable
- but you should really use github

<https://github.com/fbaptiste/python-deepdive>

To follow along you will therefore need:

CPython 3.6 or higher

Jupyter Notebook

Your favorite Python editor: VSCode, PyCharm, command line + VIM/Nano/...

I use Anaconda's Python installation: <https://conda.io/docs/index.html>

SEQUENCES

INTRODUCTION

what are sequences?

indexing starting at 0

slices include lower bound index, but exclude upper bound index

slicing

slice objects

modifying mutable sequences

copying sequences – shallow and deep

implementing custom sequence types

sorting

list comprehensions

SEQUENCE TYPES

Copyright © 2014

What is a sequence?

In Math: $S = x_1, x_2, x_3, x_4, \dots$ (countable sequence)

Note the sequence of indices: 1, 2, 3, 4, ...

We can refer to any item in the sequence by using its index number x_2 or $S[2]$

So we have a concept of the first element, the second element, and so on... → positional ordering

Python lists have a concept of positional order, but sets do not

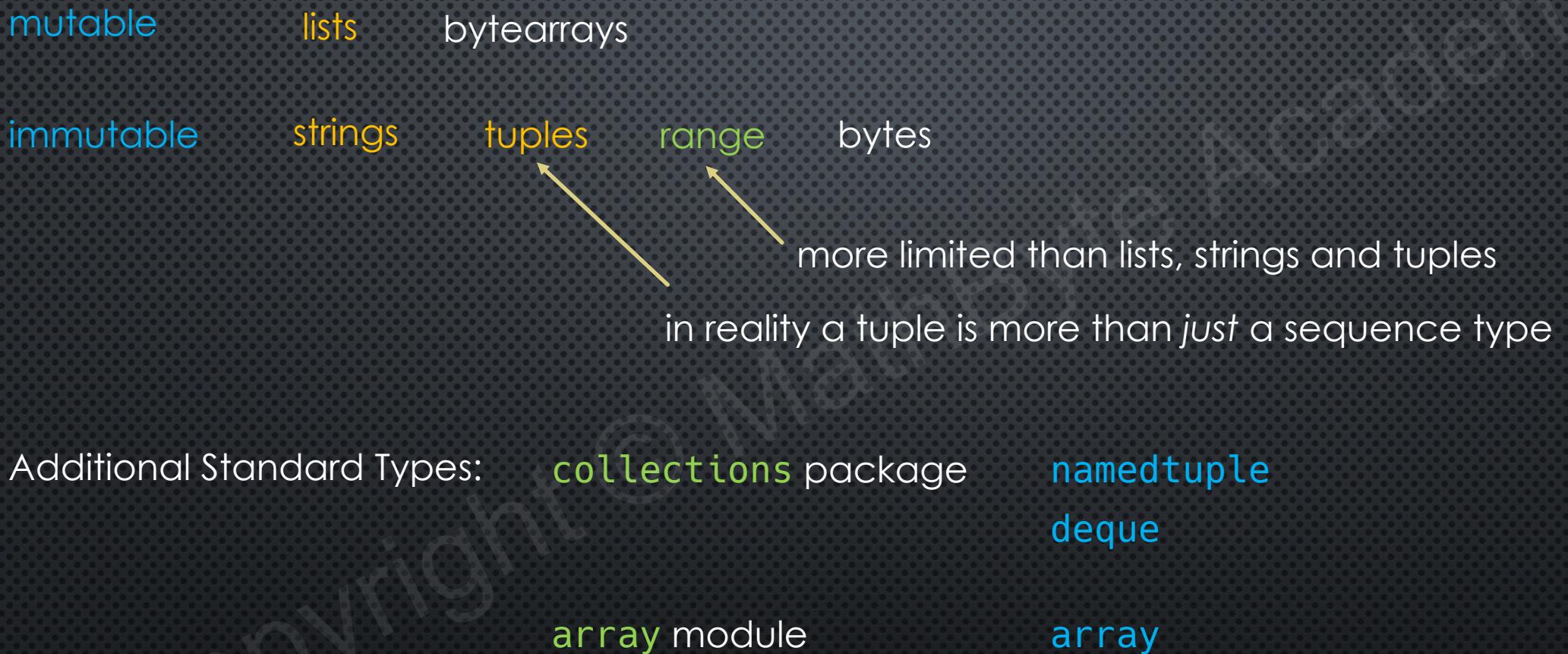
A list is a sequence type

A set is not

In Python, we start index numbers at 0, not 1 (we'll see why later)

$S = x_0, x_1, x_2, x_3, \dots$ → $S[2]$ is the third element

Built-In Sequence Types



Homogeneous vs Heterogeneous Sequences

Strings are **homogeneous** sequences

each element is of the **same** type (a character) `'python'`

Lists are **heterogeneous** sequences

each element may be a **different** type `[1, 10.5, 'python']`

Homogeneous sequence types are usually more efficient (storage wise at least)

e.g. prefer using a **string** of characters, rather than a **list** or **tuple** of characters

Iterable Type vs Sequence Type

What does it mean for an object to be **iterable**?

it is a **container** type of object and we can list out the elements in that object **one by one**

So any sequence type **is** iterable

```
l = [1, 2, 3]
```

```
for e in l  
    l[0]
```



But an iterable **is not necessarily** a sequence type → iterables are **more general**

```
s = {1, 2, 3}
```

```
for e in s
```



```
s[0]
```



Standard Sequence Methods

Built-in sequence types, both **mutable** and **immutable**, support the following methods

`x in s`

`s1 + s2`

concatenation

`x not in s`

`s * n` (or `n * s`)

(`n` an integer)

repetition

`len(s)`

`min(s)`

(if an ordering between elements of `s` is defined)

`max(s)`

This is not the same as the ordering (position) of elements inside the container, this is the ability to compare pairwise elements using an order comparison (e.g. `<`, `<=`, etc.)

`s.index(x)`

index of first occurrence of `x` in `s`

`s.index(x, i)`

index of first occurrence of `x` in `s` at or after index `i`

`s.index(x, i, j)`

index of first occurrence of `x` in `s` at or after index `i` and before index `j`

Standard Sequence Methods

`s[i]` the element at index `i`

`s[i:j]` the slice from index `i`, to (but not including) `j`

`s[i:j:k]` extended slice from index `i`, to (but not including) `j`, in steps of `k`

Note that slices will return in the `same` container type

We will come back to slicing in a lot more detail in an upcoming video

`range` objects are more `restrictive`:

no concatenation / repetition

`min`, `max`, `in`, `not in` not as efficient

Hashing

Immutable sequence types may support hashing `hash(s)`

but not if they contain mutable types!

We'll see this in more detail when we look at Mapping Types

Review: Beware of Concatenations

`x = [1, 2]`

`a = x + x`

`a → [1, 2, 1, 2]`

`x = 'python'`

`a = x + x`

`a → 'pythonpython'`

`x = [[0, 0]]`

`a = x + x`

`a → [[0, 0], [0, 0]]`

`id(x[0])`

`==`

`id(a[0])`

`==`

`id(a[1])`

`a[0][0] = 100`

`a → [[100, 0], [100, 0]]`

`a[0] is x[0]
a[1] is x[0]`

Review: Beware of Repetitions

```
a = [1, 2] * 2
```

```
a → [1, 2, 1, 2]
```

```
a = 'python' * 2
```

```
a → 'pythonpython'
```

```
a = [[0, 0]] * 2
```

```
a → [[0, 0], [0, 0]]
```

`id`

`==`

`id(a[0])`

`==`

`id(a[1])`

```
a[0][0] = 100    a → [[100, 0], [100, 0]]
```

Same happens here, but because strings are immutable it's quite safe

```
a = ['python'] * 2
```

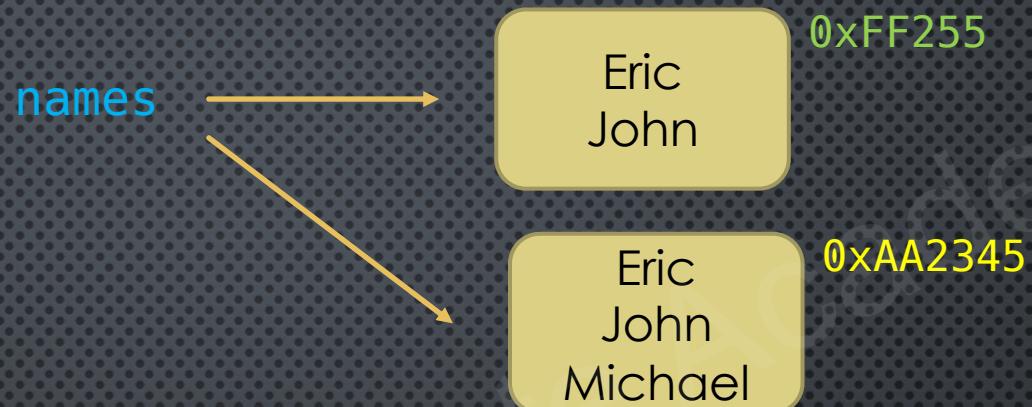
```
a → ['python', 'python']
```

MUTABLE SEQUENCE TYPES

Copyright © 2018, Packt Publishing Ltd.

Mutating Objects

```
names = ['Eric', 'John']
```



```
names = names + ['Michael']
```

This is NOT mutation!

Mutating an object means changing the object's **state** without creating a new object

```
names = ['Eric', 'John']  
names.append('Michael')
```



Mutating Using []

`s[i] = x` element at index `i` is replaced with `x`

`s[i:j] = s2` slice is replaced by the contents of the iterable `s2`

`del s[i]` removes element at index `i`

`del s[i:j]` removes entire slice

We can even assign to extended slices: `s[i:j:k] = s2`

We will come back to mutating using slicing in a lot more detail in an upcoming video

Some methods supported by mutable sequence types such as lists

`s.clear()` removes all items from `s`

`s.append(x)` appends `x` to the end of `s`

`s.insert(i, x)` inserts `x` at index `i`

`s.extend(iterable)` appends contents of `iterable` to the end of `s`

`s.pop(i)` removes and returns element at index `i`

`s.remove(x)` removes the first occurrence of `x` in `s`

`s.reverse()` does an in-place reversal of elements of `s`

`s.copy()` returns a shallow copy

and more...

INDEX BASE AND SLICE BOUNDS

RATIONALE

Valid Questions

Why does sequence indexing start at 0, and not 1?

Why does a sequence slice `s[i:j]` include `s[i]`, but exclude `s[j]`?

this is not just an arbitrary choice → there are rational and practical reasons behind doing so

We want to determine how we should handle sequences of consecutive integers

→ represent positions of elements in a sequence

`['a', 'b', 'c', 'd']`

1 2 3 4

0 1 2 3

Slice Bounds

Consider the following sequence of integers 1, 2, 3, ..., 15

How can we describe this range of numbers without using an ellipsis (...)?

- a) $1 \leq n \leq 15$
- b) $0 < n \leq 15$
- c) $1 \leq n < 16$
- d) $0 < n < 16$

(b) and (d) can become odd at times.

Suppose we want to describe the **unsigned** integers 0, 1, 2, ..., 10

Using (b) or (d) we would need to use a **s signed integer** for the lower bound:

- b) $-1 < n \leq 10$
- d) $-1 < n < 11$

Now consider this sequence: 2, 3, ..., 16

- a) $2 \leq n \leq 16$
- c) $2 \leq n < 17$

How many elements are in this sequence? 15

Calculating number of elements from bounds in (a) and (c)

$$a) 15 = 16 - 2 + 1 \quad \# = \text{upper} - \text{lower} + 1$$

$$c) 15 = 17 - 2 \quad \# = \text{upper} - \text{lower}$$

So, (c) seems simpler for that calculation

We'll get to a second reason in a bit, but for now we'll use convention (c)

Starting Indexing at 0 instead of 1

When we count elements we naturally start counting at 1, so why start indexing at 0?

Consider the following sequence:

2, 3, 4, ..., 16

sequence length: 15

index n (1 based)

1, 2, 3, ..., 15

$1 \leq n < 16$ upper bound = length + 1

index n (0 based)

0, 1, 2, ..., 14

$0 \leq n < 15$ upper bound = length

For any sequence s, the index range is given by:

0 based: $0 \leq n < \text{len}(s)$

1 based: $1 \leq n < \text{len}(s) + 1$

So, 0 based appears simpler

Another reason for choosing **0** based indexing

Consider this sequence:

	a, b, c, d, ... z
1 based	1, 2, 3, 4, ..., 26
0 based	0, 1, 2, 3, ..., 25

How many elements come **before d**? 3 elements

1 based	index(d) → 4	4-1 elements
0 based	index(d) → 3	3 elements

So, using **0** based indexing, the number of elements that precede an element at some index

→ is the index itself

Summarizing so far...

choosing **0** based indexing for sequences

describing ranges of indices using **range(l, u)** → $l \leq n < u$

we have the following results

the indices of any sequence **s** are given by: **range(0, len(s))** $[0 \leq n < \text{len}(s)]$

first index: **0** last index: **len(s)-1**

number of indices before index **n**: **n**

the length of a **range(l, u)** is given by: **l - u**

s = [a, b, c, ..., z] **len(s) → 26**

indices → **range(0, 26)**

n elements precede **s[n]**

Slices

Because of the conventions on starting indexing at **0** and defining ranges using **[lower , upper)**

we can think of slicing in these terms:

inclusive
exclusive

Each item in a sequence is like a box, with the indices **between** the boxes:



First 2 elements: $s[0:2]$ $s[:2]$

Everything else: $s[2:6]$ $s[2:]$

In general we can split a sequence into two
with k elements in the first subsequence:

$s[:k]$ $s[k:]$

COPYING SEQUENCES

Copyright © 2014 Pearson Education, Inc.

Why copy sequences?

Mutable sequences can be modified.

Sometimes you want to make sure that whatever sequence you are working with cannot be modified, either inadvertently by yourself, or by 3rd party functions

We saw an example of this earlier with list concatenations and repetitions.

Also consider this example:

```
def reverse(s):  
    s.reverse()  
    return s
```



```
s = [10, 20, 30]
```

```
new_list = reverse(s)
```

```
new_list → [30, 20, 10]
```

```
s → [30, 20, 10]
```

We should have passed it a copy of our list if we did not intend for our original list to be modified

Soapbox

```
def reverse(s):  
    s.reverse()  
    return s
```

Generally we write functions that do not modify the contents of their arguments.

But sometimes we really want to do so, and that's perfectly fine → **in-place** methods

However, to clearly indicate to the caller that something is happening in-place, we should **not** return the object we modified

If we don't return **s** in the above example, the caller will probably wonder why not?

So, in this case, the following would be a better approach:

```
def reverse(s):  
    s.reverse()
```

and if we do not do in-place reversal, then we return the reversed sequence

```
def reverse(s):  
    s2 = <copy of s>  
    s2.reverse()  
    return s2
```

How to copy a sequence

We can copy a sequence using a variety of methods: `s = [10, 20, 30]`

Simple Loop

```
cp = []
for e in s:
    cp.append(e)
```

definitely non-Pythonic!

List Comprehension `cp = [e for e in s]`

The copy method `cp = s.copy()` (not implemented in immutable types, such as tuples or strings)

Slicing

`cp = s[0:len(s)]` or, more simply `cp = s[:]`

The `copy` module

```
list()           list_2 = list(list_1)
```

Note: `tuple_2 = tuple(tuple_1)` and `t[:]` does not create a new tuple!

Watch out when copying entire immutable sequences

```
l1 = [1, 2, 3]
```

```
l2 = list(l1)          l2 → [1, 2, 3]      id(l1) ≠ id(l2)
```

```
t1 = (1, 2, 3)
```

```
t2 = tuple(t1)        t2 → (1, 2, 3)      id(t1) = id(t2)    same object!
```

```
t1 = (1, 2, 3)
```

```
t2 = t1[:]            t2 → (1, 2, 3)      id(t1) = id(t2)    same object!
```

Same thing with strings, also an immutable sequence type

Since the sequence is **immutable**, it is actually **OK** to return the same sequence

Shallow Copies

Using any of the techniques above, we have obtained a copy of the original sequence

```
s = [10, 20, 30]
cp = s.copy()
cp[0] = 100          cp → [100, 20, 30]  s → [10, 20, 30]
```

Great, so now our sequence `s` will always be safe from unintended modifications? Not quite...

```
s = [[10, 20], [30, 40]]
cp = s.copy()
cp[0] = 'python'      cp → ['python', [30, 40]]  s → [[10, 20], [30, 40]]

cp[1][0] = 100

cp → ['python', [100, 40]]  s → [[10, 20], [100, 40]]
```

Shallow Copies

What happened?

When we use any of the copy methods we saw a few slides ago, the copy essentially copies all the **object references** from one sequence to another

`s = [a, b]` `id(s) → 1000` `id(s[0]) → 2000` `id(s[1]) → 3000`

`cp = s.copy()` `id(cp) → 5000` `id(cp[0]) → 2000` `id(cp[1]) → 3000`

When we made a copy of `s`, the sequence was **copied**, but it's **elements** point to the same memory address as the **original** sequence **elements**

The sequence was **copied**, but it's **elements were not**

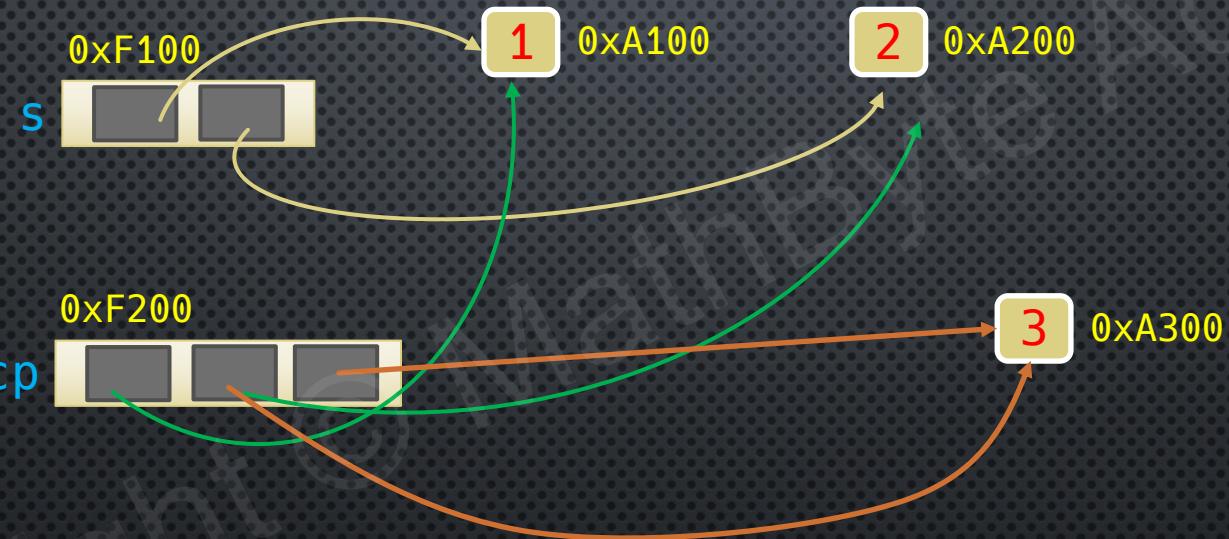
This is called a **shallow copy**

Shallow Copies

```
s = [ 1, 2 ]  
cp = s.copy()
```

```
cp.append(3)
```

```
cp[1] = 3
```



If the elements of `s` are immutable, such as integers in this example, then not really important

Shallow Copies

But, if the elements of `s` are **mutable**, then it can be important

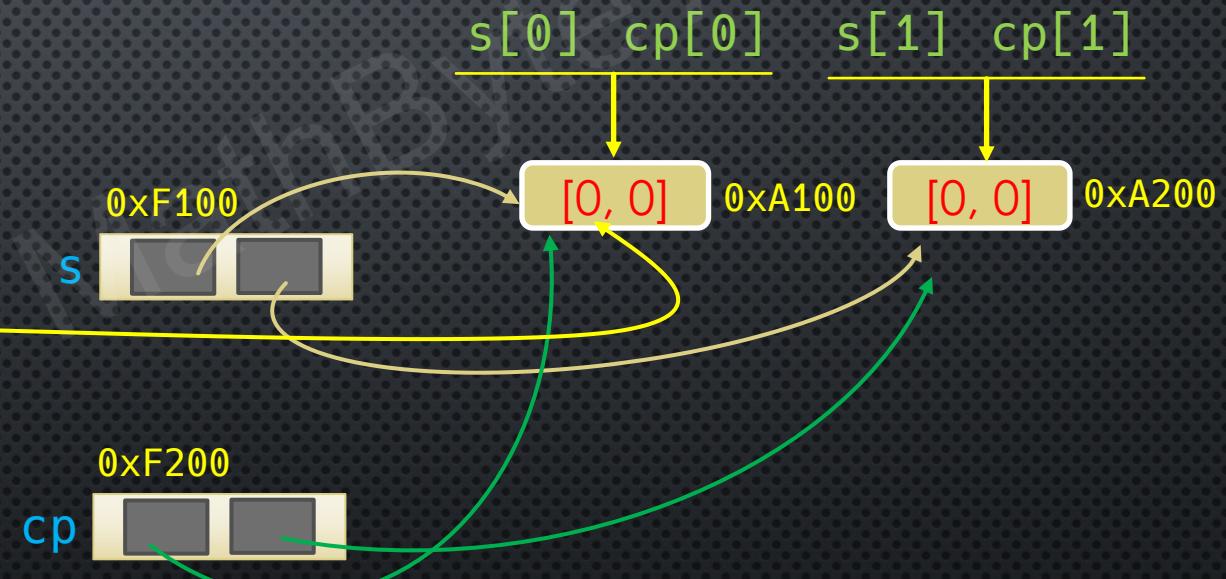
```
s = [ [0, 0], [0, 0] ]
```

```
cp = s.copy()
```

```
cp[0][0] = 100
```

```
cp → [ [100, 0], [0, 0] ]
```

```
s → [ [100, 0], [0, 0] ]
```



Deep Copies

So, if collections contain **mutable** elements, shallow copies are not sufficient to ensure the copy can never be used to modify the original!

Instead, we have to do something called a **deep copy**.

For the previous example we might try this:

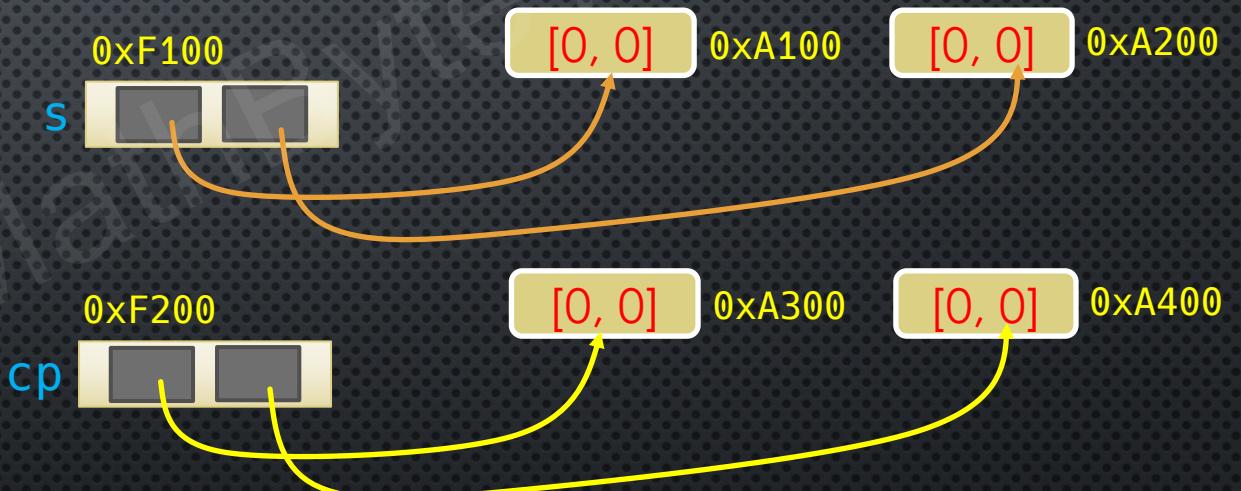
```
s = [ [0, 0], [0, 0] ]  
cp = [e.copy() for e in s]
```

In this case:

`cp` is a copy of `s`

but also, **every** element of `cp` is a **copy** of the corresponding element in `s`

↑
shallow copy



Deep Copies

But what happens if the mutable elements of `s` themselves contain mutable elements?

```
s = [ [ 0, 1, 2, 3 ], [ 4, 5, 6, 7 ] ]
```



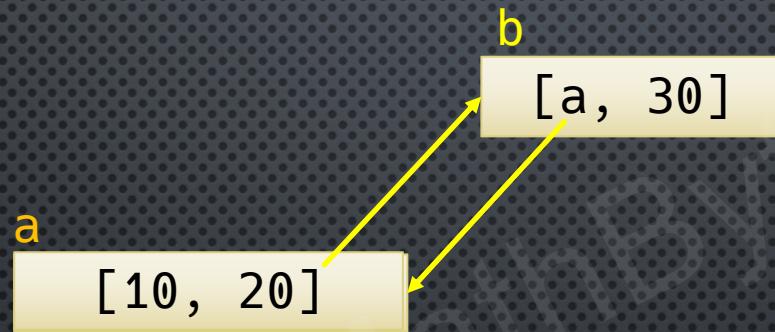
We would need to make copies at least 3 levels deep to ensure a true deep copy

Deep copies, in general, tend to need a **recursive** approach

Deep Copies

Deep copies are not easy to do. You might even have to deal with circular references

```
a = [10, 20]  
b = [a, 30]  
a.append(b)
```



If you wrote your own deep copy algorithm, you would need to handle this circular reference!

Deep Copies

In general, objects know how to make shallow copies of themselves
built-in objects like lists, sets, and dictionaries do - they have a `copy()` method

The standard library `copy` module has generic `copy` and `deepcopy` operations

The `copy` function will create a shallow copy

The `deepcopy` function will create a deep copy, handling nested objects, and circular references properly

Custom classes can implement the `__copy__` and `__deepcopy__` methods to allow you to override how shallow and deep copies are made for your custom objects

We'll revisit this advanced topic of overriding deep copies of custom classes in the OOP series of this course.

Deep Copies

Suppose we have a custom class as follows:

```
def MyClass:  
    def __init__(self, a):  
        self.a = a
```

```
from copy import copy, deepcopy
```

```
x = [10, 20]
```

```
obj = MyClass(x)
```

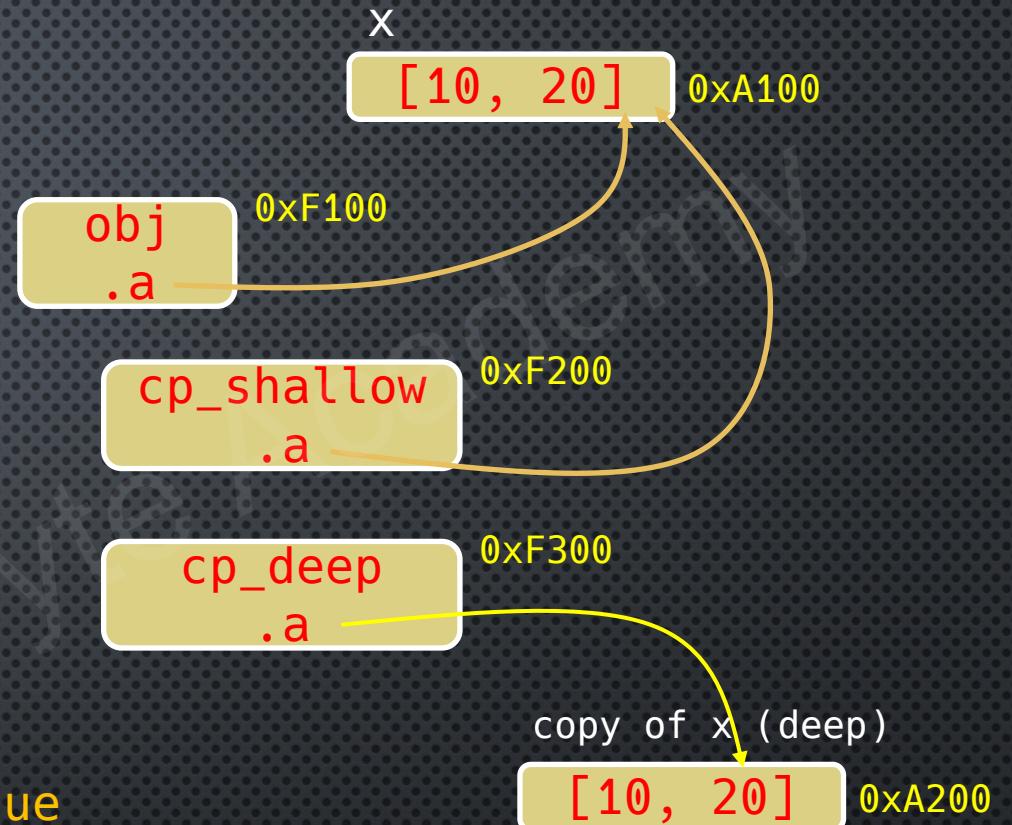
```
cp_shallow = copy(obj)
```

```
cp_deep = deepcopy(obj)
```

`x is obj.a → True`

`cp_shallow.a is obj.a → True`

`cp_deep.a is obj.a → False`



Deep Copies

```
def MyClass:  
    def __init__(self, a):  
        self.a = a  
  
x = MyClass(500)  
y = MyClass(x)          y.a is x → True  
lst = [x, y]
```

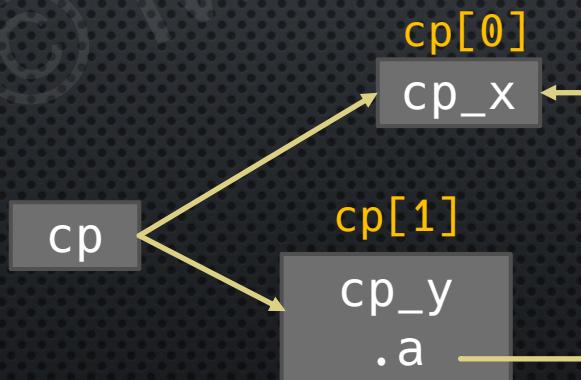
```
cp = deepcopy(lst)
```

```
cp[0] is x → False  
cp[1] is y → False  
cp[1].a is x → False
```

```
cp[1].a is cp[0] → True
```



this is **not** a circular reference
but there is a **relationship**
between `y.a` and `x`



relationship between `cp_y.a` and `cp_x`
is maintained!

SLICING

Copyright © 2014

© 2014 Pearson Education, Inc.

We've used slicing in this course before, but now it's time to dive deeper into slicing

Slicing relies on indexing → only works with sequence types

Mutable Sequence Types

extract data

assign data

Example

```
l = [1, 2, 3, 4, 5]
```

```
l[0:2] = ('a', 'b', 'c')
```

```
l[0:2] → ['a', 'b', 'c', 3, 4, 5]
```

Immutable Sequence Types

extract data

The Slice Type

Although we usually slice sequences using the more conventional notation:

```
my_list[i:j]
```

slice definitions are actually objects → of type `slice`

```
s = slice(0, 2)
```

```
type(s) → slice
```

```
s.start → 0
```

```
s.end → 2
```

```
l = [1, 2, 3, 4, 5]
```

```
l[s] → [1, 2]
```

This can be useful because we can name slices and use symbols instead of a literal subsequently

Similar to how you can name ranges in Excel...

Slice Start and Stop Bounds

[*i:j*]

start at *i* (including *i*) stop at *j* (excluding *j*)

all integers *k* where *i* \leq *k* < *j*

also remember that indexing is zero-based

It can be convenient to think of slice bounds this way



[1:4]

Effective Start and Stop Bounds

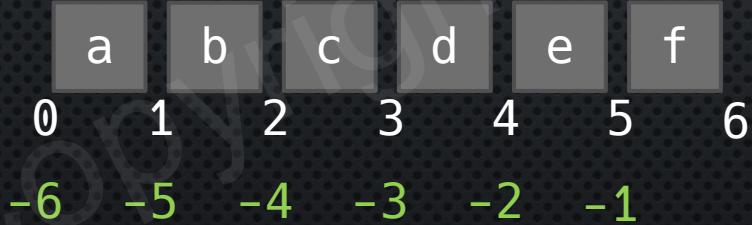
Interestingly the following works:

```
l = ['a', 'b', 'c', 'd', 'e', 'f']  
l[3:100] → ['d', 'e', 'f'] No error!
```

we can specify slices that are "out of bounds"

In fact, negative indices work too:

```
l[-1] → 'f'  
l[-3: -1] → ['d', 'e']
```



Step Value

Slices also support a third argument – the **step** value

[**i:j:k**]

(a.k.a stride)

`slice(i, j, k)`

When not specified, the step value defaults to **1**

```
l = [0      1      2      3      4      5  
  'a', 'b', 'c', 'd', 'e', 'f'  
 -6     -5     -4     -3     -2     -1]
```

`l[0:6:2]` 0, 2, 4 \rightarrow ['a', 'c', 'e']

`l[1:6:3]` 1, 4 \rightarrow ['b', 'e']

`l[1:15:3]` 1, 4 \rightarrow ['b', 'e']

`l[-1:-4:-1]` -1, -2, -3 \rightarrow ['f', 'e', 'd']

Range Equivalence

Any slice essentially defines a sequence of indices that is used to select elements for another sequence

In fact, any indices defined by a slice can also be defined using a range

The difference is that slices are defined independently of the sequence being sliced

The equivalent range is only calculated once the length of the sequence being sliced is known

Example

[0:100]	sequence of length 10	→ range(0, 10)
	sequence of length 6	→ range(0, 6)

Transformations `[i:j]`

The effective indices "generated" by a slice are actually dependent on the length of the sequence being sliced

Python does this by reducing the slice using the following rules:

`seq[i:j]`

`l = ['a', 'b', 'c', 'd', 'e', 'f']`
length = 6

if `i > len(seq)` → `len(seq)`

`[0:100] → range(0, 6)`

if `j > len(seq)` → `len(seq)`

if `i < 0` → `max(0, len(seq) + i)`

`[-10:3] → range(0, 3)`

if `j < 0` → `max(0, len(seq) + j)`

`[-5:3] → range(1, 3)`

`i` omitted or `None` → `0`

`[:100] → range(0, 6)`

`j` omitted or `None` → `len(seq)`

`[3:] → range(3, 6)`
`[:] → range(0, 6)`

Transformations $[i:j:k]$, $k > 0$

With extended slicing things change depending on whether k is negative or positive

$$[i:j:k] = \{x = i + n * k \mid 0 \leq n < (j-i)/k\}$$

$k > 0$ the indices are: $i, i+k, i+2k, i+3k, \dots, < j$ stopping when j is reached or exceeded, but never including j itself

$l = [\begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 'a', 'b', 'c', 'd', 'e', 'f' \end{matrix}]$
-6 -5 -4 -3 -2 -1 length = 6

if $i, j > \text{len(seq)}$ $\rightarrow \text{len(seq)}$ $[0:100:2] \rightarrow \text{range}(0, 6, 2)$

if $i, j < 0$ $\rightarrow \max(0, \text{len(seq)} + i/j)$ $[-10:100:2] \rightarrow \text{range}(0, 6, 2)$

i omitted or **None** $\rightarrow 0$ $[-5:100:2] \rightarrow \text{range}(1, 6, 2)$

j omitted or **None** $\rightarrow \text{len(seq)}$ $[:6:2] \rightarrow \text{range}(0, 6, 2)$

j omitted or **None** $\rightarrow \text{len(seq)}$ $[1::2] \rightarrow \text{range}(1, 6, 2)$

j omitted or **None** $\rightarrow \text{len(seq)}$ $[:2] \rightarrow \text{range}(0, 6, 2)$

so same rules as $[i:j]$ – makes sense, since that would be the same as $[i:j:1]$

Transformations `[i:j:k]`, $k < 0$

`[i:j:k] = {x = i + n * k | 0 <= n < (j-i)/k}`

$k < 0$ the indices are: $i, i+k, i+2k, i+3k, \dots, > j$

`l = ['a', 'b', 'c', 'd', 'e', 'f']`
 $\begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \\ -6 & -5 & -4 & -3 & -2 & -1 \end{matrix}$

`length = 6`

`if i, j > len(seq) → len(seq) - 1`

`[5:2:-1] → range(5, 2, -1)`

`[10:2:-1] → range(5, 2, -1)`

`if i, j < 0 → max(-1, len(seq) + i/j)`

`[5:-2:-1] → range(5, 4, -1)`

`[-2:-5:-1] → range(4, 1, -1)`

`[-2:-10:-1] → range(4, -1, -1)`

`i omitted or None → len(seq) - 1`

`[:-2:-1] → range(5, 4, -1)`

`j omitted or None → -1`

`[5::-1] → range(5, -1, -1)`

`[::-1] → range(5, -1, -1)`

Summary

[$i:j]$ [$i:j:k]$ $k > 0$ [$i:j:k]$ $k < 0$

$i > \text{len(seq)}$	len(seq)	$\text{len(seq)} - 1$
$j > \text{len(seq)}$	len(seq)	$\text{len(seq)} - 1$
$i < 0$	$\max(0, \text{len(seq)} + i)$	$\max(-1, \text{len(seq)} + i)$
$j < 0$	$\max(0, \text{len(seq)} + j)$	$\max(-1, \text{len(seq)} + j)$
i omitted / <code>None</code>	0	$\text{len(seq)} - 1$
j omitted / <code>None</code>	len(seq)	-1

Examples

```
l = [ 'a', 'b', 'c', 'd', 'e', 'f' ]  
    -6      -5      -4      -3      -2      -1
```

length = 6

`[-10:10:1]` $-10 \rightarrow 0$

$10 \rightarrow 6$

$\rightarrow \text{range}(0, 6)$

`[10:-10:-1]` $10 \rightarrow 5$

$-10 \rightarrow \max(-1, 6-10) \rightarrow \max(-1, -4) \rightarrow -1$

$\rightarrow \text{range}(5, -1, -1)$

We can of course easily define empty slices!

`[3:-1:-1]` $3 \rightarrow 3$

$-1 \rightarrow \max(-1, 6-1) \rightarrow 5$

$\rightarrow \text{range}(3, 5, -1)$

Example

seq = sequence of length 6

seq[::-1] i is omitted → len(seq) - 1 → 5
 j is omitted → -1

 → range(5, -1, -1) → 5, 4, 3, 2, 1, 0

seq = 'python'

seq[::-1] → 'nohtyp'

If you get confused...

The `slice` object has a method, `indices`, that returns the equivalent range start/stop/step for any slice given the length of the sequence being sliced:

```
slice(start, stop, step).indices(length) → (start, stop, step)
```

the values in this tuple can be used to generate a list of indices using the `range` function

```
slice(10, -5, -1)    with a sequence of length 6
```

```
i=10 > 6 → 6-1 → 5
```

```
j=-5 < 0 → max(-1, 6+-5) → max(-1, 1) → 1      → range(5, 1, -1)  
                                                → 5, 4, 3, 2
```

```
slice(10, -5, -1).indices(6) → (5, 1, -1)
```

```
list(range(*slice(10,-5,-1).indices(6))) → [5, 4, 3, 2]
```

CUSTOM SEQUENCE TYPES

PART 1

Creating our own Sequence types

We will cover Abstract Base Classes later in this course, so we'll revisit this topic again

At its most basic, an immutable sequence type should support two things:

returning the **length** of the sequence (*technically, we don't even really need that!*)

given an **index**, returning the element at that index

If an object provides this functionality, then we should in theory be able to:

retrieve elements **by index** using square brackets **[]**

iterate through the elements using Python's native looping mechanisms

e.g. for loops, comprehensions

How Python does it

Remember that sequence types are iterables, but not all iterables are sequence types

Sequence types, at a minimum, implement the following methods:

`__len__` `__getitem__`

At its most basic, the `__getitem__` method takes in a single integer argument – the index

However, it may also choose to handle a `slice` type argument

So how does this help when iterating over the elements of a sequence?

The `__getitem__` method

The `__getitem__` method should return an element of the sequence based on the specified index

or raise an `IndexError` exception if the index is out of bounds

(and may, but does not have to, support negative indices and slicing)

Python's `list` object implements the `__getitem__` method:

```
my_list = ['a', 'b', 'c', 'd', 'e', 'f']

my_list.__getitem__(0) → 'a'
my_list.__getitem__(1) → 'b'
my_list.__getitem__(-1) → 'f'

my_list.__getitem__(slice(None, None, -1))

→ ['f', 'e', 'd', 'c', 'b', 'a']
```

The `__getitem__` method

But if we specify an index that is out of bounds:

```
my_list.__getitem__(100) → IndexError
```

```
my_list.__getitem__(-100) → IndexError
```

All we really need from this `__getitem__` method is the ability to

return an element for a valid index

raise an `IndexError` exception for an invalid index

Also remember, that sequence indices start at 0

i.e. we always know the index of the first element of the sequence

Implementing a `for` loop

So now we know: sequence indexing starts at `0`

`__getitem__(i)` will return the element at index `i`

`__getitem__(i)` will raise an `IndexError` exception when `i` is out of bounds

```
my_list = [0, 1, 2, 3, 4, 5]
```

```
for item in my_list:    index = 0
    print(item ** 2)
                    while True:
                        try:
                            item = my_list.__getitem__(index)
                        except IndexError:
                            break
                        print(item ** 2)
                        index += 1
```

The point is that if the object implements `__getitem__`

we can iterate through it using a `for` loop, or even a comprehension

The `__len__` Method

In general sequence types support the Python built-in function `len()`

To support this all we need to do is implement the `__len__` method in our custom sequence type

```
my_list = [0, 1, 2, 3, 4, 5]
```

```
len(my_list) → 6
```

```
my_list.__len__() → 6
```

Writing our own Custom Sequence Type

to implement our own custom sequence type we should then implement:

`__len__`

`__getitem__`

At the very least `__getitem__` should:

return an element for a valid index `[0, length-1]`

raise an `IndexError` exception if index is out of bounds

Additionally we can choose to support:

negative indices $i < 0 \rightarrow i = length - i$

slicing handle `slice` objects as argument to `__getitem__`

IN-PLACE CONCATENATION AND REPETITION

Copyright © 2014

Concatenation +

Let's use Python's `list` as an example

We can concatenate two lists together by using the `+` operator

This will **create a new** list combining the elements of both lists

```
l1 = [1, 2, 3]           id(l1) = 0xFFFF100
```

```
l2 = [4, 5, 6]           id(l2) = 0xFFFF200
```

```
l1 = l1 + l2 → [1, 2, 3, 4, 5, 6]   id(l1) = 0xFFFF300
```

In-Place Concatenation $+=$

Recall that for numbers I have said many times that

`a = a + 10` and `a += 10` meant the same thing?

That's true for numbers... but not in general!

it's true for numbers, strings, tuples \rightarrow in general, true for immutable types

but not lists!

`l1 = [1, 2, 3]` `id(l1) = 0xFFFF100`

`l2 = [4, 5, 6]` `id(l2) = 0xFFFF200`

`l1 += l2` \rightarrow `[1, 2, 3, 4, 5, 6]` `id(l1) = 0xFFFF100`

the list was mutated

In-Place Concatenation `+=`

For immutable types, such as number, strings, tuples the behavior is different

`t += t1` has the same effect as `t = t + t1`

Since `t` is immutable, `+=` does **NOT** perform in-place concatenation

Instead it creates a **new** tuple that concatenates the two tuples and returns the **new object**

`t1 = (1, 2, 3)` `id(t1) = 0xFFFF100`

`t2 = (4, 5, 6)` `id(t2) = 0xFFFF200`

`t1 += t2` → `(1, 2, 3, 4, 5, 6)` `id(t1) = 0xFFFF300`

In-Place Repetition $\ast=$

Similar result hold for the \star and $\ast=$ operator

`l1 = [1, 2, 3]` `id(l1) = 0xFFFF100`

`l1 = l1 * 2` \rightarrow `[1, 2, 3, 1, 2, 3]` `id(l1) = 0xFFFF200`

But the in-place repetition operator works this way:

`l1 = [1, 2, 3]` `id(l1) = 0xFFFF100`

`l1 *= 2` \rightarrow `[1, 2, 3, 1, 2, 3]` `id(l1) = 0xFFFF100`

the list was **mutated**

ASSIGNMENTS IN MUTABLE SEQUENCES

Copyright ©

Assigning Values via Indexes, Slices and Extended Slices

We have seen how we can extract elements from a sequence by using indexing, slicing, and extended slicing

[i]

[i:j] slice(i, j)

[i:j:k] slice (i, j, k) k ≠ 1 (if k=1 then it's just a standard slice)

Mutable sequences support assignment via a specific index

and they also support assignment via slices

The value being assigned via slicing and extended slicing must to be an iterable
(any iterable, not just a sequence type)

Replacing a Slice

A slice can be replaced with another iterable

For regular slices (non-extended), the slice and the iterable need not be the same length

`l = [1, 2, 3, 4, 5]` `l[1:3] → [2, 3]`

`l[1:3] = (10, 20, 30)` `l → [1, 10, 20, 30, 4, 5]`

The list `l` was mutated → `id(l)` did not change

With extended slicing, the extended slice and the iterable must have the same length

`l = [1, 2, 3, 4, 5]` `l[0:4:2] → [1, 3]`

`l[0:4:2] = [10, 30]` `l → [10, 2, 30, 4, 5]`

The list `l` was mutated

Deleting a Slice

Deletion is really just a special case of replacement

We simply assign an empty iterable

→ works for standard slicing only
(extended slicing replacement needs same length)

```
l = [1, 2, 3, 4, 5]
```

```
l[1:3] → [2, 3]
```

```
l[2:3] = []
```

```
l → [1, 4, 5]
```

The list `l` was mutated

Insertions using Slices

We can also insert elements using slice assignment

The trick here is that the slice must be empty
otherwise it would just replace the elements in the slice

```
l = [1, 2, 3, 4, 5]
```

```
l[1:1] → []
```

```
l[1:1] = 'abc'
```

```
l[1:1] → [1, 'a', 'b', 'c', 2, 3, 4, 5]
```

The list `l` was mutated

Obviously this will also not work with extended slices

extended slice assignment requires both lengths to be the same

but for insertion we need the slice to be empty,
and the iterable to have some values

CUSTOM SEQUENCES

PART 2

Concatenation and In-Place Concatenation

When dealing with the `+` and `+=` operators in the context of sequences we usually expect them to mean **concatenation**

But essentially, it is just an overloaded definition of these operators

We can overload the definition of these operators in our custom classes by using the methods:

`--add--` `--iadd--`

In general (but not necessarily), we expect:

`obj1 + obj2` → `obj1` and `obj2` are of the same type
→ result is a new object also of the same type

`obj1 += obj2` → `obj2` is any iterable
→ result is the original `obj1` memory reference
(i.e. `obj1` was mutated)

Repetition and In-Place Repetition

When dealing with the `*` and `*=` operators in the context of sequences we usually expect them to mean repetition

But essentially, it is just an overloaded definition of these operators

We can overload the definition of these operators in our custom classes by using the methods:

`__mul__` `__imul__`

In general (but not necessarily), we expect:

- | | |
|------------------------|--|
| <code>obj1 * n</code> | <ul style="list-style-type: none">→ <code>n</code> is a non-negative integer→ result is a new object of the same type as <code>obj1</code> |
| <code>obj1 *= n</code> | <ul style="list-style-type: none">→ <code>n</code> is a non-negative integer→ result is the original <code>obj1</code> memory reference<ul style="list-style-type: none">(i.e. <code>obj1</code> was mutated) |

Assignment

We saw in an earlier lecture how we can implement accessing elements in a custom sequence type

- `__getitem__` → `seq[n]`
- `seq[i:j]`
- `seq[i:j:k]`

We can handle assignments in a very similar way, by implementing

`__setitem__`

There a few restrictions with assigning to slices that we have already seen (at least with lists):

For any slice we could only assign an iterable

For extended slices only, both the slice and the iterable must have the same length

Of course, since we are implementing `__setitem__` ourselves, we could technically make it do whatever we want!

Additional Sequence Functions and Operators

There are other operators and functions we can support:

<code>__contains__</code>	<code>in</code>
<code>__delitem__</code>	<code>del</code>
<code>__rmul__</code>	<code>n * seq</code>

The way Python works is that when it encounters an expression such as:

	<code>a + b</code>	<code>a * b</code>
it first tries	<code>a.__add__(b)</code>	<code>a.__mul__(b)</code>

if `a` does not support the operation (`TypeError`), it then tries:

<code>b.__radd__(a)</code>	<code>b.__rmul__(a)</code>
----------------------------	----------------------------

Implementing `append`, `extend`, `pop`

Actually there's nothing special going here.

If we want to, we can just implement methods of the same name (not `special` methods)

and they can just behave the same way as we have seen for lists for example

SORTING SEQUENCES

Copyright © 2014

Sorting and Sort Keys

Sorting a sequence of numbers is something easily understood

But we do have to consider the **direction** of the sort: **ascending** **descending**

Python provides a **sorted()** function that will sort a given iterable

The default sort direction is **ascending**

The **sorted()** function has an optional keyword-only argument called **reverse** which defaults to **False**

If we set it to **True**, then the sort will sort in **descending** order

But one really important thing we need to think about: **ordering**

→ obvious when sorting **real numbers**

Sorting and Sort Keys

What about non-numerical values?

'a', 'b', 'c'

'A', 'a', 'B', 'b', 'C', 'c'

'hello', 'python', 'bird', 'parrot'

(0, 0) (1, 1) (2, 2)

(0, 0) (0, 1) (1, 0)

rectangle_1, rectangle_2, rectangle_3



strings are comparable so this is still OK
although is 'a' < 'A' or 'a' > 'A' or 'a' == 'A'

↑
True

When items are pairwise comparable (< or >
we can use that ordering to sort

but what happens when they are not?

→ Sort Keys

Sorting and Sort Keys

'b', 'x', 'a'

ASCII character codes

a → 97
b → 98
x → 120

ord('a') → 97

We now associate the ASCII numerical value with each character, and sort based on that value

items	'b'	'x'	'a'		'a'	'b'	'x'	
keys	98	120	97	→	97	98	120	
	'B'	'b'	'A'	'a'	'X'	'x'	'1'	'?'
	66	98	65	97	88	120	49	63
→	'1'	'?'	'A'	'B'	'X'	'a'	'b'	'x'
	49	63	65	66	88	97	98	120

You'll note that the sort keys have a natural sort order

Sorting and Sort Keys

Let's say we want to sort a list of `Person` objects based on their age

`p1.age → 30`

`p2.age → 15`

`p3.age → 5`

`p4.age → 32`

item	p1	p2	p3	p4	→	p3	p2	p1	p4
keys	30	15	5	32		5	15	30	32

We could also generate the key value, for any given person, using a function

```
def key(p):  
    return p.age
```

```
key = lambda p: p.age
```

```
sort [p1, p2, p3, p4]
```

using sort keys generated by the function `key = lambda p: p.age`

(assumes the `Person` class has an age property)

Sorting and Sort Keys

The sort keys need not be numerical → they just need to have a natural sort order (< or >)

item 'hello' 'python' 'parrot' 'bird'

keys 'o' 'n' 't' 'd' ← last character of each string

→ 'bird' 'python' 'hello' 'parrot'
 'd' 'n' 'o' 't'

key = lambda s: s[-1]

Python's `sorted` function

That's exactly what Python's `sorted` function allows us to do

Optional keyword-only argument called `key`

if provided, `key` must be a function that for any given element in the sequence being sorted returns the sort key

The sort key does not have to be numerical

→ it just needs to be values that are themselves pairwise comparable (such as `<` or `>`)

If `key` is not provided, then Python will sort based on the natural ordering of the elements

i.e. they must be pairwise comparable (`<`, `>`)

If the elements are not pairwise comparable, you will get an exception

Python's `sorted` function

```
sorted(iterable, key=None, reverse=False)
```



keyword-only

The `sorted` function:

- makes a **copy** of the iterable
- returns the sorted elements in a **list**
- uses a sort algorithm called **TimSort**
- a **stable** sort

→ named after Tim Peters Python 2.3, 2002

<https://en.wikipedia.org/wiki/Timsort>

Side note: for the "natural" sort of elements, we can always think of the keys as the elements themselves

```
sorted(iterable) ←→ sorted(iterable, key=lambda x: x)
```

Stable Sorts

A stable sort is one that maintains the relative order of items that have equal keys
(or values if using natural ordering)

```
p1.age → 30  
p2.age → 15  
p3.age → 5  
p4.age → 32  
p5.age → 15
```

```
sorted((p1, p2, p3, p4, p5), key=lambda p: p.age)
```

→ [p3 p2 p5 p1 p4]



keys equal

p2 preceded p5 in original tuple

→ p2 precedes p5 in sorted list

In-Place Sorting

If the iterable is mutable, **in-place** sorting is **possible**

But that will depend on the particular type you are dealing with

Python's **list** objects support in-place sorting

The list class has a **sort()** instance method that does in-place sorting

```
l = [10, 5, 3, 2]      id(l) → 0xFF42
```

```
l.sort()
```

```
l → [2, 3, 5, 10]      id(l) → 0xFF42
```

Compared to **sorted()**

- same **TimSort** algorithm
- same keyword-only arg: **key**
- same keyword-only arg: **reverse** (default is **False**)
- **in-place** sorting, does not copy the data
- only works on **lists** (it's a method in the **list** class)

LIST COMPREHENSIONS

Copyright © 2014
DataCamp

Quick Recap

You should already know what list comprehensions are, but let's quickly recap their syntax and how they work:

goal → generate a list by **transforming**, and optionally **filtering**, another **iterable**

- start with some iterable
- create empty new list
- iterate over the original iterable
- skip over certain values (filter)
- transform value and append to new list

```
other_list = ['this', 'is', 'a', 'parrot']

new_list = []

for item in other_list:
    if len(item) > 2:
        new_list.append(item[::-1])
```

List comprehension:

```
new_list = [item[::-1] for item in other_list if len(item) > 2]
```

transformation

iteration

filter

Formatting the Comprehension Expression

If the comprehension expression gets too long, it can be split over multiple lines

For example, let's say we want to create a list of squares of all the integers between **1** and **100** that are not divisible by 2, 3 or 5

```
sq = [i**2 for i in range(1, 101) if i%2 and i%3 and i%5]
```

We could write this over multiple lines:

```
sq = [i**2  
      for i in range(1, 101)  
      if i%2 and i%3 and i%5]
```

Comprehension Internals

Comprehensions have their own local scope – just like a function

We should think of a list comprehension as being wrapped in a function that is created by Python that will return the new list when executed

`sq = [i**2 for i in range(10)]`

RHS

When the RHS is compiled: Python creates a temporary function that will be used to evaluate the comprehension

```
def temp():
    new_list = []
    for i in range(10):
        new_list.append(i**2)
    return new_list
```

When the line is executed:
 Executes `temp()`
 Stores the returned object (the list) in memory
 Points `sq` to that object

We'll disassemble some Python code in the coding video to actually see this

Comprehension Scopes

So comprehensions are basically functions

They have their own local scope:

```
[item ** 2 for item in range(100)]
```

local symbol

But they can access global variables:

```
# module1.py
```

```
num = 100
```

global symbol

```
sq = [item**2 for item in range(num)]
```

local symbol

As well as nonlocal variables:

```
def my_func(num):
```

```
    sq = [item**2 for item in range(num)]
```

nonlocal symbol

Closures!!

Nested Comprehensions

Comprehensions can be nested within each other

And since they are functions, a nested comprehension can access (nonlocal) variables from the enclosing comprehension!

```
[ [i * j for j in range(5)] for i in range(5)]
```

nested comprehension

outer comprehension

local variable: **i**

closure

local variable: **j**

free variable: **i**

Nested Loops in Comprehensions

We can have nested loops (as many levels as we want) in comprehensions.

This is not the same as nested comprehensions

```
l = []
for i in range(5):
    for j in range(5):
        for k in range(5):
            l.append((i, j, k))
```

```
l = [(i, j, k) for i in range(5) for j in range(5) for k in range(5)]
```

Note that the **order** in which the for loops are specified in the comprehension correspond to the order of the nested loops

Nested Loops in Comprehensions

Nested loops in comprehensions can also contain **if** statements

Again the order of the **for** and **if** statements does matter, just like a normal set of **for** loops and **if** statements

```
l = []
for i in range(5):
    for j in range(5):
        if i==j:
            l.append((i, j))
```

```
l = []
for i in range(5):
    if i==j:
        for j in range(5):
            l.append((i, j))
```

won't work!

```
l = [(i, j) for i in range(5) for j in range(5) if i == j]
```

```
l = [(i, j) for i in range(5) if i == j for j in range(5)]
```

won't work!

j is created here

j is referenced after
it has been created

Nested Loops in Comprehensions

```
l = []
for i in range(1, 6):
    if i%2 == 0:
        for j in range(1, 6):
            if j%3 == 0:
                l.append((i,j))
```

```
[(i, j)
 for i in range(1, 6) if i%2==0
 for j in range(1, 6) if j%3==0]
```



```
l = []
for i in range(1, 6):
    for j in range(1, 6):
        if i%2==0:
            if j%3 == 0:
                l.append((i,j))
```

```
[(i, j)
 for i in range(1, 6)
 for j in range(1, 6)
 if i%2==0
 if j%3==0]
```



```
l = []
for i in range(1, 6):
    for j in range(1, 6):
        if i%2==0 and j%3==0:
            l.append((i,j))
```

```
[(i, j)
 for i in range(1, 6)
 for j in range(1, 6)
 if i%2==0 and j%3==0]
```



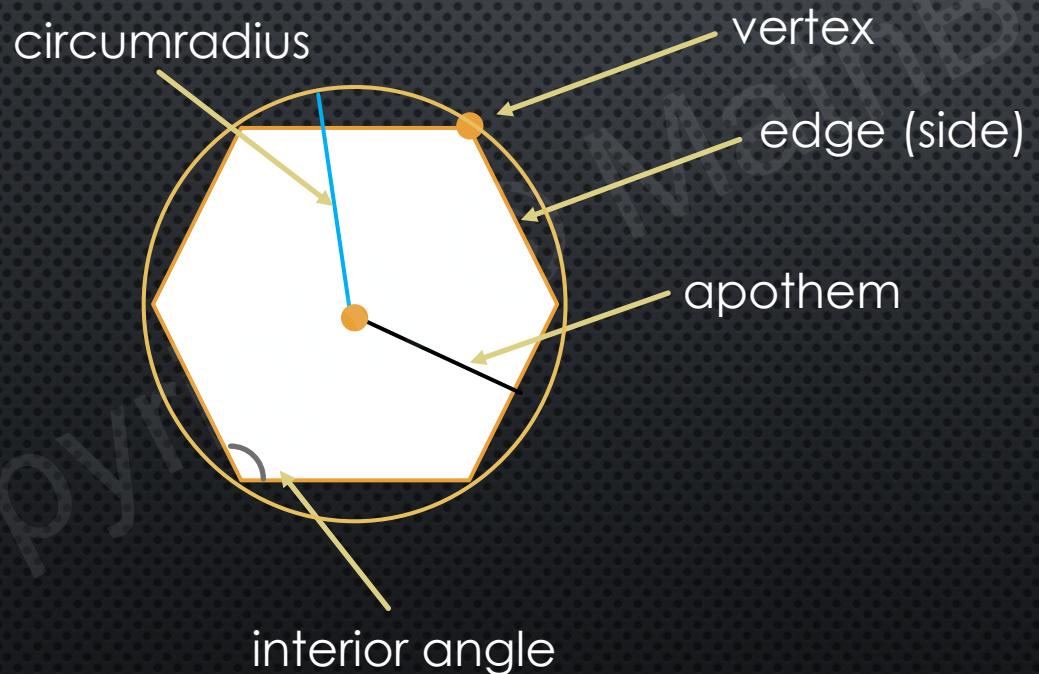
PROJECT

Copyright © 2014

Background Information

A regular strictly convex polygon is a polygon that has the following characteristics:

- all interior angles are less than 180°
- all sides have equal length



Background Information

For a regular strictly convex polygon with

- n edges (= n vertices)
- R circumradius

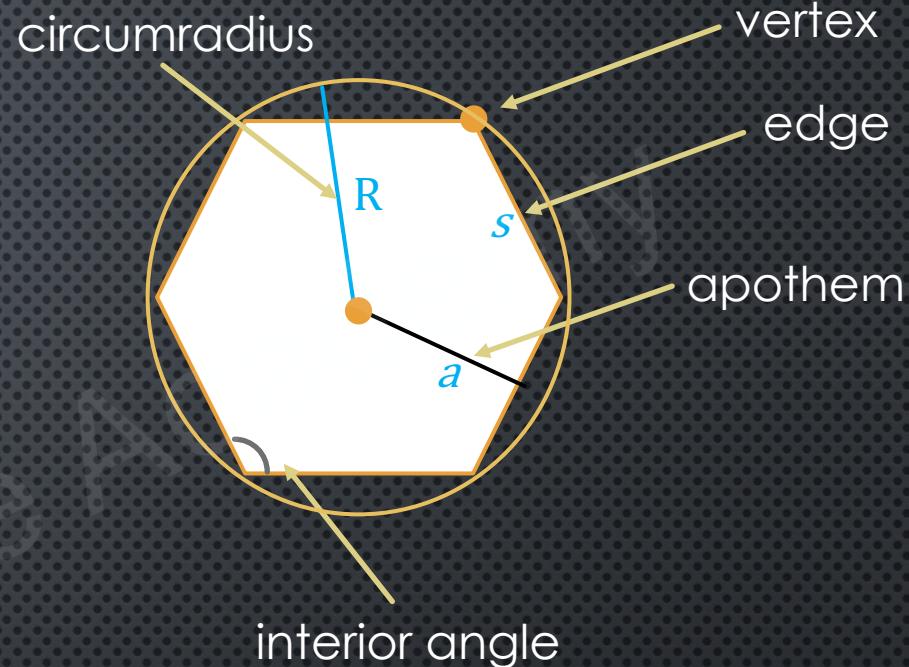
$$\text{interior angle} = (n - 2) \times \frac{180}{n}$$

$$\text{edge length } s = 2 R \sin\left(\frac{\pi}{n}\right)$$

$$\text{apothem } a = R \cos\left(\frac{\pi}{n}\right)$$

$$\text{area} = \frac{1}{2} n s a$$

$$\text{perimeter} = n s$$



Goal 1

Create a `Polygon` class:

Initializer

- number of edges/vertices
- circumradius

Properties

- # edges
- # vertices
- interior angle
- edge length
- apothem
- area
- perimeter

Functionality

- a proper representation (`__repr__`)
- implements equality (`==`) based on # vertices and circumradius (`__eq__`)
- implements `>` based on number of vertices only (`__gt__`)

Goal 2

Implement a **Polygons** sequence type:

Initializer

- number of vertices for largest polygon in the sequence
- common circumradius for all polygons

Properties

- max efficiency polygon: returns the Polygon with the highest **area : perimeter** ratio

Functionality

- functions as a sequence type (`__getitem__`)
- supports the `len()` function (`__len__`)
- has a proper representation (`__repr__`)

ITERABLES AND ITERATORS

INTRODUCTION

What is an iterable?

Something fit for iterating over



→ we'll see a more formal definition for Python's iterable **protocol**

Already seen: Sequences and iteration

More general concept of iteration

Iterators

→ get next item, no indexes needed

→ consumables

Iterables

Consuming iterators manually

Relationship between sequence types and iterators

Infinite Iterables

Lazy Evaluation

Iterator Delegation

ITERATING COLLECTIONS

Copyright © 2014

Iterating Sequences

- We saw that in the last section → `__getitem__`
- assumed indexing started at `0`
- iteration: `__getitem__(0)`, `__getitem__(1)`, etc

But iteration can be more **general** than based on **sequential indexing**

All we need is:

- a bucket of items → **collection, container**
- get **next** item → no concept of ordering needed
→ just a way to get items out of the container one by one
a specific order in which this happens is not required – but can be

Example: Sets

Sets are unordered collections of items

```
s = {'x', 'y', 'b', 'c', 'a'}
```

Sets are not indexable

```
s[0]
```

→ TypeError - 'set' object does not support indexing

But sets are iterable

```
for item in s:  
    print(item)
```

→
y
c
x
b
a

Note that we have no idea of the order in
which the elements are returned in the
iteration

The concept of next

For general iteration, all we really need is the concept of "get the next item" in the collection

If a collection object implements a `get_next_item` method

we can get elements out of the collection, one after the other, this way:

```
get_next_item( )  
get_next_item( )  
get_next_item( )
```

and we could iterate over the collection as follows:

```
for _ in range(10):  
    item = coll.get_next_item()  
    print(item)
```

But how do we know when to stop asking for the next item?

i.e. when all the elements of the collection have been returned by calling `get_next_item()`?

→ `StopIteration` built-in Exception

Attempting to build an Iterable ourselves

Let's try building our own class, which will be a collection of squares of integers

We could make this a sequence, but we want to avoid the concept of indexing

In order to implement a next method, we need to know what we've already "handed out" so we can hand out the "next" item without repeating ourselves

```
class Squares:  
    def __init__(self):  
        self.i = 0  
  
    def next_(self):  
        result = self.i ** 2  
        self.i += 1  
        return result
```

Iterating over Squares

```
sq = Squares()
```

```
for _ in range(5):  
    item = sq.next_()  
    print(item)
```

→ 0
 1
 4
 9
 16

There are a few issues:

- the collection is essentially infinite
- cannot use a **for** loop, comprehension, etc
- we cannot restart the iteration "from the beginning"

```
class Squares:  
    def __init__(self):  
        self.i = 0  
  
    def next_(self):  
        result = self.i ** 2  
        self.i += 1  
        return result
```

Refining the Squares Class

we first tackle the idea of making the collection finite

- we specify the `size` of the collection when we `create` the instance
- we raise a `StopIteration` exception if `next_` has been called too many times

```
class Squares:  
    def __init__(self):  
        self.i = 0  
  
    def next_(self):  
        result = self.i ** 2  
        self.i += 1  
        return result
```

```
class Squares:  
    def __init__(self, length):  
        self.i = 0  
        self.length = length  
  
    def next_(self):  
        if self.i >= self.length:  
            raise StopIteration  
        else:  
            result = self.i ** 2  
            self.i += 1  
            return result
```

Iterating over **Squares** instances

```
sq = Squares(5)      create a collection of length 5
```

```
while True:          start an infinite loop
```

```
    try:  
        item = sq.next_()    try getting the next item  
        print(item)
```

```
    except StopIteration:  catch the StopIteration exception → nothing left to iterate  
        break                break out of the infinite while loop – we're done iterating
```

Output:

```
0  
1  
4  
9  
16
```

```
class Squares:  
    def __init__(self, length):  
        self.i = 0  
        self.length = length  
  
    def next_(self):  
        if self.i >= self.length:  
            raise StopIteration  
        else:  
            result = self.i ** 2  
            self.i += 1  
            return result
```

Python's `next()` function

Remember Python's `len()` function?

We could implement that function for our custom type by implementing the special method: `__len__`

Python has a built-in function: `next()`

We can implement that function for our custom type by implementing the special method: `__next__`

```
class Squares:  
    def __init__(self, length):  
        self.i = 0  
        self.length = length  
  
    def __next__(self):  
        if self.i >= self.length:  
            raise StopIteration  
        else:  
            result = self.i ** 2  
            self.i += 1  
            return result
```

Iterating over `Squares` instances

```
sq = Squares(5)
while True:
    try:
        item = next(sq)
        print(item)
    except StopIteration:
        break
```

Output:

0
1
4
9
16

We still have some issues:

- cannot iterate using `for` loops, comprehensions, etc
- once the iteration starts we have no way of re-starting it
 - and once all the items have been iterated (using `next`) the object becomes useless for iteration → **exhausted**

ITERATORS

Copyright © 2014

DATA
STRUCTURES
AND
ALGORITHMS

Where we're at so far...

We created a custom container type object with a `__next__` method

But it had several drawbacks:

- cannot use a `for` loop
- once we start using `next` there's no going back
- once we have reached `StopIteration` we're basically done with the object

Let's tackle the `loop` issue first

We saw how to iterate using `__next__`, `StopIteration`, and a `while` loop

This is actually how Python handles `for` loops in general

Somehow, we need to tell Python that our class has that `__next__` method and that it will behave in a way consistent with using a `while` loop to iterate

Python knows we have `__next__`, but how does it know we implement `StopIteration`?

The iterator Protocol

A protocol is simply a fancy way of saying that our class is going to implement certain functionality that Python can count on

To let Python know our class can be iterated over using `__next__` we implement the iterator protocol

The iterator protocol is quite simple – the class needs to implement two methods:

→ `__iter__` this method should just return the object (class instance) itself

sounds weird, but we'll understand why later



→ `__next__` this method is responsible for handing back the next element from the collection and raising the `StopIteration` exception when all elements have been handed out

An object that implements these two methods is called an iterator

Iterators

An **iterator** is therefore an object that implements:

`__iter__` → just returns the object itself

`__next__` → returns the next item from the container, or raises **StopIteration**

If an object is an iterator, we can use it with **for** loops, comprehensions, etc

Python will know how to loop (iterate) over such an object
(basically using the same **while** loop technique we used)

Example

Let's go back to our **Squares** example, and make it into an iterator

```
class Squares:  
    def __init__(self, length):  
        self.i = 0  
        self.length = length  
  
    def __next__(self):  
        if self.i >= self.length:  
            raise StopIteration  
        else:  
            result = self.i ** 2  
            self.i += 1  
            return result  
  
    def __iter__(self):  
        return self
```

```
sq = Squares(5)  
for item in sq: →  
    print(item)  
0  
1  
4  
9  
16
```

Still one issue though!

The iterator cannot be "restarted"

Once we have looped through all the items
the iterator has been **exhausted**

To loop a **second** time through the
collection we have to create a **new**
instance and loop through that

ITERATORS AND ITERABLES

Copyright © 2018

Iterators

We saw that an **iterator** is an object that implements

`__iter__` → returns the object itself

`__next__` → returns the next element

The drawback is that iterators get **exhausted** → become useless for iterating again
→ become throw away objects

But two distinct things going on:

maintaining the collection of items (the container) (e.g. creating, mutating (if mutable), etc)

iterating over the collection

Why should we have to re-create the collection of items just to iterate over them?

Separating the Collection from the Iterator

Instead, we would prefer to separate these two

Maintaining the data of the collection should be one object

Iterating over the data should be a separate object → iterator

That object is throw-away → but we don't throw away the collection

The collection is iterable

but the iterator is responsible for iterating over the collection

The iterable is created once

The iterator is created every time we need to start a fresh iteration

Example

```
class Cities:  
    def __init__(self):  
        self._cities = ['Paris', 'Berlin', 'Rome', 'London']  
        self._index = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self._index >= len(self._cities):  
            raise StopIteration  
        else:  
            item = self._cities[self._index]  
            self._index += 1  
            return item
```

Cities instances are iterators

Every time we want to run a new loop, we have to create a new instance of Cities

This is wasteful, because we should not have to re-create the _cities list every time

Example So, let's separate the object that maintains the cities, from the iterator itself

```
class Cities:  
    def __init__(self):  
        self._cities = ['New York', 'New Delhi', 'Newcastle']  
  
    def __len__(self):  
        return len(self._cities)  
  
class CityIterator:  
    def __init__(self, cities):  
        self._cities = cities  
        self._index = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self._index >= len(self._cities):  
            raise StopIteration  
        else:  
            etc...
```

Example

To use the `Cities` and `CityIterator` together here's how we would proceed:

```
cities = Cities()      create an instance of the container object  
  
city_iterator = CityIterator(cities)    create a new iterator – but see how we pass in the  
                                         existing cities instance  
  
for city in cities_iterator:  
    print(city)                      can now use the iterator to iterate
```

At this point, the `cities_iterator` is exhausted

If we want to re-iterate over the collection, we need to create a new one

```
city_iterator = CityIterator(cities)  
  
for city in cities_iterator:  
    print(city)
```

But this time, we did not have to re-create the collection – we just passed in the existing one!

So far...

At this point we have:

- a container that maintains the collection items

- a separate object, the iterator, used to iterate over the collection

So we can iterate over the collection as many times as we want

we just have to **remember** to create a **new iterator** every time

It would be nice if we did not have to do that manually every time

and if we could just iterate over the **Cities** object instead of **CityIterator**

This is where the formal definition of a Python **iterable** comes in...

Iterables

An **iterable** is a Python object that implements the **iterable protocol**

The **iterable protocol** requires that the object implement a single method

`__iter__` returns a **new instance** of the iterator object used to iterate over the iterable

```
class Cities:  
    def __init__(self):  
        self._cities = ['New York', 'New Delhi', 'Newcastle']  
  
    def __len__(self):  
        return len(self._cities)  
  
    def __iter__(self):  
        return CityIterator(self)
```

Iterable vs Iterator

An **iterable** is an object that implements

`__iter__` → returns an **iterator** (in general, a new instance)

An **iterator** is an object that implements

`__iter__` → returns itself (an iterator) (not a new instance)

`__next__` → returns the next element

So iterators are themselves iterables

but they are iterables that become exhausted

Iterables on the other hand never become exhausted

because they always return a new iterator that is then used to iterate

Iterating over an iterable

Python has a built-in function `iter()`

It calls the `__iter__` method
(we'll actually come back to this for sequences!)

The first thing Python does when we try to iterate over an object

it calls `iter()` to obtain an iterator

then it starts iterating (using `next`, `StopIteration`, etc)

using the iterator returned by `iter()`

LAZY ITERABLES

Lazy Evaluation

This is often used in class properties

properties of classes may not always be populated when the object is created

value of a property only becomes known when the property is requested - deferred

Example

```
class Actor:  
    def __init__(self, actor_id):  
        self.actor_id = actor_id  
        self.bio = lookup_actor_in_db(actor_id)  
        self.movies = None  
  
    @property  
    def movies(self):  
        if self.movies is None:  
            self.movies = lookup_movies_in_db(self.actor_id)  
        return self.movies
```

Application to Iterables

We can apply the same concept to certain iterables

We do not calculate the next item in an iterable until it is actually requested

Example

iterable → `Factorial(n)`

will return factorials of consecutive integers from `0` to `n-1`

do not pre-compute all the factorials

wait until `next` requests one, then calculate it

This is a form of lazy evaluation

Application to Iterables

Another application of this might be retrieving a list of forum posts

Posts might be an iterable

each call to **next** returns a list of 5 posts (or some page size)

but uses **lazy loading**

→ every time **next** is called, go back to database and get next 5 posts

Application to Iterables → Infinite Iterables

Using that lazy evaluation technique means that we can actually have **infinite** iterables

Since items are not computed until they are requested

we can have an infinite number of items in the collection



Don't try to use a for loop over such an iterable
unless you have some type of exit condition in your loop
→ otherwise infinite loop!

Lazy evaluation of iterables is something that is used a lot in Python!

We'll examine that in detail in the next section on generators

THE `iter()` FUNCTION

Copyright © 2014

What happens when Python performs an iterationon over an iterable?

The very first thing Python does is call the `iter()` function on the object we want to iterate

If the object implements the `__iter__` method, that method is called
and Python uses the returned iterator

What happens if the object does not implement the `__iter__` method?

Is an exception raised immediately?

Sequence Types

So how does iterating over a **sequence** type – that maybe only implemented `__getitem__` work?

I just said that Python always calls `iter()` first

You'll notice I did not say Python always calls the `__iter__` method

I said it calls the `iter()` function!!

In fact, if `obj` is an object that only implements `__getitem__`

`iter(obj)` → returns an **iterator** type object!

Some form of magic at work?

Not really!

Let's think about sequence types and how we can iterate over them

Suppose `seq` is some sequence type that implements `__getitem__` (but not `__iter__`)

Remember what happens when we request an index that is out of bounds from the
`__getitem__` method? → `IndexError`

```
index = 0
while True:
    try:
        print(seq[index])
        index += 1
    except IndexError:
        break
```

Making an Iterator to iterate over any Sequence

This is basically what we just did!

```
class SeqIterator:  
    def __init__(self, seq):  
        self.seq = seq  
        self.index = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__:  
        try:  
            item = self.seq[self.index]  
            self.index += 1  
            return item  
        except IndexError:  
            raise StopIteration()
```

Calling `iter()`

So when `iter(obj)` is called:

Python first looks for an `__iter__` method

- if it's there, use it
- if it's not
 - look for a `__getitem__` method
 - if it's there create an iterator object and return that
 - if it's not there, raise a `TypeError` exception (not iterable)

Testing if an object is iterable

Sometimes (very rarely!)

you may want to know if an object is iterable or not

But now you would have to check if they implement

`__getitem__` or `__iter__`

and that `__iter__` returns an iterator

Easier approach:

```
try:  
    iter(obj)  
except TypeError:  
    # not iterable  
    <code>  
else:  
    # is iterable  
    <code>
```

PYTHON'S BUILT-IN ITERABLES AND ITERATORS

Copyright © 2018, DataCamp

Python provides many functions that return iterables or iterators

Additionally, the iterators perform **lazy** evaluation

You should always be aware of whether you are dealing with an iterable or an iterator

why?

- if an object is an **iterable** (but **not** an **iterator**) you can iterate over it many times
- if an object is an **iterator** you can iterate over it only once

`range(10)` → iterable

`zip(l1, l2)` → iterator

`enumerate(l1)` → iterator

`open('cars.csv')` → iterator

`dictionary .keys()` → iterable

`dictionary .values()` → iterable

`dictionary .items()` → iterable

and many more...

ITERATING CALLABLES

Copyright © 2014

Iterating over the return values of a callable

Consider a callable that provides a countdown from some start value:

```
countdown( ) → 5  
countdown( ) → 4  
countdown( ) → 3  
countdown( ) → 2  
countdown( ) → 1  
countdown( ) → 0  
countdown( ) → -1  
...
```

We now want to run a loop that will call `countdown()` until `0` is reached

We could certainly do that using a loop and testing the value to break out of the loop once `0` has been reached

```
while True:  
    val = countdown()  
    if val == 0:  
        break  
    else:  
        print(val)
```

An iterator approach

We could take a different approach, using **iterators**, and we can also make it quite generic

Make an iterator that knows two things:

- the **callable** that needs to be called

- a value (the **sentinel**) that will result in a **StopIteration** if the callable returns that value

The iterator would then be implemented as follows:

when **next()** is called:

- call the callable and get the result

- if the **result** is equal to the **sentinel** → **StopIteration**

- and "exhaust" the iterator

- otherwise **return** the result

We can then simply iterate over the iterator until it is exhausted

The first form of the `iter()` function

We just studied the first form of the `iter()` function:

`iter(iterable)` → iterator for iterable

if the iterable did not implement the `iterator protocol`, but implemented the `sequence protocol`

`iter()` creates a iterator for us (leveraging the sequence protocol)

Notice that the `iter()` function was able to generate an iterator for us automatically

The second form of the `iter()` function

`iter(callable, sentinel)`

This will return an `iterator` that will:

call the callable when `next()` is called

and either raise `StopIteration` if the `result` is `equal` to the `sentinel` value
or return the `result` otherwise

REVERSED ITERATION

Copyright © 2014

NCSC-Navy

Iterating a sequence in reverse order

If we have a sequence type, then iterating over the sequence in reverse order is quite simple:

```
for item in seq[::-1]:  
    print(item)
```

This works, but is **wasteful** because it makes a copy of the sequence

```
for i in range(len(seq)):  
    print(seq[len(seq) - i - 1])
```

This is more efficient, but the syntax is messy

```
for i in range(len(seq)-1, -1, -1):  
    print(seq[i])
```

This is cleaner and just as efficient, because it creates an **iterator** that will iterate backwards over the sequence – it does not copy the data like the first example

Both `__getitem__` and `__len__` must be implemented

We can override how reversed works by implementing the `__reversed__` special method

Iterating an iterable in reverse

Unfortunately, `reversed()` will not work with custom iterables without a little bit of extra work

When we call `reversed()` on a custom iterable, Python will look for and call the `__reversed__` function

That function should `return` an `iterator` that will be used to perform the reversed iteration

So basically we have to implement a reverse iterator ourselves

Just like the `iter()` method, when we call `reversed()` on an object:

looks for and calls `__reversed__` method

if it's not there, uses `__getitem__` and `__len__`
to create an iterator for us

exception otherwise

Card Deck Example

In the code exercises I am going to build an iterable containing a deck of 52 sorted cards

2 Spades ... Ace Spades, 2 Hearts ... Ace Hearts, 2 Diamonds ... Ace Diamonds, 2 Clubs ... Ace Clubs

But I don't want to create a list containing all the pre-created cards → Lazy evaluation

So I want my iterator to figure out the suit and card name for a given index in the sorted deck

```
SUITS = ['Spades', 'Hearts', 'Diamonds', 'Clubs']
```

```
RANKS = [2, 3, ..., 10, 'J', 'Q', 'K', 'A']
```

We assume the deck is sorted as follows:

iterate over SUITS

for each suit iterate over RANKS

card = combination of suit and rank

Card Deck Example

```
SUITS = ['Spades', 'Hearts', 'Diamonds', 'Clubs']

RANKS = [2, 3, ..., 10, 'J', 'Q', 'K', 'A']

2S ... AS 2H ... AH 2D ... AD 2C ... AC
```

There are `len(SUITS)` suits 4

There are `len(RANKS)` ranks 13

The deck has a length of: `len(SUITS) * len(RANKS)` 52

Each `card` in this deck has a `positional index`: a number from 0 to `len(deck) - 1` 0 - 51

To find the `suit index` of a card at index `i`:

```
i // len(RANKS)
```

Examples

5th card (6S) → index 4

```
→ 4 // 13 → 0
```

16th card (4H) → index 15

```
→ 15 // 13 → 1
```

To find the `rank index` of a card at index `i`:

```
i % len(RANKS)
```

Examples

5th card (6S) → index 4

```
→ 4 % 13 → 4
```

16th card (4H) → index 15

```
→ 15 % 13 → 2
```

GENERATORS

Copyright © 2014
McGraw-Hill Education

generators

→ a type of iterator

generator functions

→ generator factories

→ they return a generator when called

→ they are not a generator themselves

generator expressions

→ uses comprehension syntax

→ a more concise way of creating generators

→ like list comprehensions, useful for simple situations

performance considerations

YIELDING AND GENERATORS

Copyright © 2014

Iterators review

Let's recall how we would write a simple iterator for factorials

```
class FactIter:  
    def __init__(self, n):  
        self.n = n  
        self.i = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self.i >= self.n:  
            raise StopIteration  
        else:  
            result = math.factorial(self.i)  
            self.i += 1  
            return result
```

Now that's quite a bit of work for a simple iterator!

There has to be a better way...

What if we could do something like this instead:

```
def factorials(n):
    for i in range(n):
        emit factorial(i)
        pause execution here
        wait for resume
    return 'done!'
```

and in our code we would want to do something like this maybe:

```
facts = factorials(4)
get_next(facts) → 0!
get_next(facts) → 1!
get_next(facts) → 2!
get_next(facts) → 3!
get_next(facts) → done!
```

Of course, getting `0!`, `1!`, `2!`, `3!` followed by a string is odd

And what happens if we call `get_next` again?

Maybe we should consider raising an exception... `StopIteration`?

And instead of calling `get_next`, why not just use `next`?

But what about that `emit`, `pause`, `resume`? → `yield`

Yield to the rescue...

The **yield** keyword does exactly what we want:

- it **emits** a value

- the function is effectively **suspended** (but it retains its current state)

- calling **next** on the function **resumes** running the function right after the yield statement

- if function **returns** something instead of yielding (finishes running) → **StopIteration** exception

```
def song():
    print('line 1')
    yield "I'm a lumberjack and I'm OK"
    print('line 2')
    yield 'I sleep all night and I work all day'
```

`lines = song()` → no output!

`line = next(lines)` → 'line 1' is printed in console
`line` → "I'm a lumberjack and I'm OK"

`line = next(lines)` → 'line 2' is printed in console
`line` → "I sleep all night and I work all day"

`line = next(lines)` → **StopIteration**

Generators

A function that uses the **yield** statement, is called a **generator function**

```
def my_func():      my_func is just a regular function  
    yield 1          calling my_func() returns a generator object  
    yield 2  
    yield 3
```

We can think of functions that contain the **yield** statement as **generator factories**

The generator is created by Python when the function is **called** → `gen = my_func()`

The resulting generator is executed by calling **next()** → `next(gen)`

the function body will execute until it encounters a **yield** statement

it **yields** the value (as **return** value of `next()`) then it **suspends** itself

until **next** is called again → suspended function **resumes** execution

if it encounters a **return** before a **yield**

→ **StopIteration** exception occurs

(Remember that if a function terminates without an explicit return, Python essentially returns a None value for us)

Generators

```
def my_func():  
    yield 1  
    yield 2  
    yield 3
```

gen = my_func() → gen is a generator

next(gen) → 1

next(gen) → 2

next(gen) → 3

next(gen) → StopIteration

Generators

`next StopIteration`

This should remind you of **iterators**!

In fact, generators **are** iterators

→ they implement the **iterator protocol**

`__iter__` `__next__`

```
def my_func():
    yield 1
    yield 2
    yield 3
```

→ they are **exhausted** when function **returns** a value

→ **StopIteration** exception

→ return value is the exception **message**

```
gen = my_func()
```

`gen.__iter__()` → `iter(gen)` → returns `gen` itself

`gen.__next__()` → `next(gen)`

Example

```
class FactIter:  
    def __init__(self, n):  
        self.n = n  
        self.i = 0  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self.i >= self.n:  
            raise StopIteration  
        else:  
            result = math.factorial(self.i)  
            self.i += 1  
            return result  
  
fact_iter = FactIter(5)
```

```
def factorials(n):  
    for i in range(n):  
        yield math.factorial(i)  
  
fact_iter = factorials(5)
```

Generators

Generator functions are functions which contain at least one `yield` statement

When a generator function is called, Python `creates` a `generator` object

Generators implement the `iterator protocol`

Generators are inherently `lazy` iterators (and can be infinite)

Generators `are` iterators, and can be used in the same way (for loops, comprehensions, etc)

Generators become `exhausted` once the function `returns` a value

MAKING AN ITERABLE FROM A GENERATOR

Copyright © 2018, DataCamp

Generators become exhausted

Generator functions are functions that use `yield`

A generator function is a `generator factory` → they `return` a (new) `generator` when called

Generators are iterators

- they can become exhausted (consumed)
- they cannot be "restarted"

This can lead to bugs if you try to iterate twice over a generator

Example

```
def squares(n):  
    for i in range(n):  
        yield i ** 2
```

sq = squares(5) → sq is a new generator (iterator)

l = list(sq) l → [0, 1, 4, 9, 16]

and sq has been exhausted

l = list(sq) l → []

Example

This of course can lead to unexpected behavior sometimes...

```
def squares(n):  
    for i in range(n):  
        yield i ** 2
```

```
sq = squares(5)
```

```
enum1 = enumerate(sq)      enumerate is lazy → hasn't iterated through sq yet
```

```
next(sq)      → 0
```

```
next(sq)      → 1
```

```
list(enum1)  → [(0,4), (1, 9), (2, 16)]
```

↑
notice how enumerate started at $i=2$

and the index value returned by enumerate is 0 , not 2

Making an Iterable

This behavior is no different than with any other iterator

As we saw before, the solution is to create an **iterable** that returns a new iterator every time

```
def squares(n):
    for i in range(n):
        yield i ** 2
```

```
sq = Squares(n)
```

```
l1 = list(sq)
```

```
l2 = list(sq)
```

```
class Squares:
    def __init__(self, n):
        self.n = n
    def __iter__(self):
        return squares(n)
```

new instance of
the generator

$l1 \rightarrow [0, 1, 4, 9, 16]$

$l2 \rightarrow [0, 1, 4, 9, 16]$

GENERATOR EXPRESSIONS

Copyright © 2014

Comprehension Syntax

We already covered comprehension syntax when we studied list comprehensions

```
l = [i ** 2 for i in range(5)]
```

As well as more complicated syntax:

- if statements
- multiple nested loops
- nested comprehensions

```
[(i, j)
 for i in range(1, 6) if i%2==0
 for j in range(1, 6) if j%3==0]
```

```
[[i * j for j in range(5)] for i in range(5)]
```

Generator Expressions

Generator expressions use the **same** comprehension syntax → including nesting, if

but instead of using **[]**

we use **()**

`[i ** 2 for i in range(5)]`

`(i ** 2 for i in range(5))`

a **list** is returned

a **generator** is returned

evaluation is **eager**

evaluation is **lazy**

has local scope

has local scope

can access nonlocal
and global scopes

can access nonlocal
and global scopes

iterable

iterator

Resource Utilization

List comprehensions are **eager**

all objects are created right away

→ takes **longer** to **create**/return the list

→ **iteration** is **faster** (objects already created)

Generators are **lazy**

object creation is delayed until requested by `next()`

→ generator is **created**/returned **immediately**

→ **iteration** is **slower** (objects need to be created)

if you iterate through **all** the elements → time performance is about the same

if you do **not** iterate through all the elements → generator more efficient

→ **entire** collections is loaded into memory

→ only a **single** item is loaded at a time

in general, generators tend to have **less** memory overhead

YIELD FROM

Copyright © 2014
by  Inc.

Delegating to another iterator

Often we may need to delegate yielding elements to another iterator

```
file1.csv  file2.csv  file3.csv
```

```
def read_all_data():
    for file in ('file1.csv', 'file2.csv', 'file3.csv'):
        with open(file) as f:
            for line in f:
                yield line
```

The inner loop is basically just using the file iterator and yielding values directly

Essentially we are **delegating** yielding to the file iterator

Simpler Syntax

We can replace this inner loop by using a simpler syntax: `yield from`

```
def read_all_data():
    for file in ('file1.csv', 'file2.csv',
    'file3.csv'):
        with open(file) as f:
            for line in f:
                yield line
```

```
def read_all_data():
    for file in ('file1.csv', 'file2.csv',
    'file3.csv'):
        with open(file) as f:
            yield from f
```

We'll come back to `yield from`, as there is a lot more to it!

PROJECT

Copyright © 2014

Background Info

Along with this project is a data file: [nyc_parking_tickets_extract.csv](#)

Here are the first few lines of data

```
Summons Number,Plate ID,Registration State,Plate Type,Issue Date,Violation Code,Vehicle Body Type,Vehicle Make,Violation Description
4006478550,VAD7274,VA,PAS,10/5/2016,5,4D,BMW,BUS LANE VIOLATION
4006462396,22834JK,NY,COM,9/30/2016,5,VAN,CHEVR,BUS LANE VIOLATION
4007117810,21791MG,NY,COM,4/10/2017,5,VAN,DODGE,BUS LANE VIOLATION
```

- fields separated by commas
- first row contains the field names
- data rows are a mix of data types: string, date, int

Note that an end-of-line character is not visible, but it's there!

Goal 1

Your first goal is to create a **lazy iterator** that will produce a **named tuple** for each row of **data**

The contents of each tuple should be an appropriate data type (e.g. date, int, string)

You can use the **split** method for string to split on the comma

You will need to use the **strip** method to remove the end-of-line character (**\n**)

Remember, the goal is to produce a lazy iterator

- you should not be reading the entire file in memory and then processing it
- the goal is to keep the required memory overhead to a minimum

Please stick to Python's built-ins and the standard library only!

Goal 2

Calculate the number of violations by car make.

Use the lazy iterator you created in Goal 1

Use lazy evaluation whenever possible

You can choose otherwise, but I would store the make and violation counts as a dictionary

- key = car make
- value = # violations

ITERATION TOOLS

Copyright © 2014

Python has many tools for working with iterables

You should already know almost all of these:

`iter` `reversed` `next` `len` `slicing`

`zip`

`filter`

`sorted`

`enumerate`

`min` `max` `sum`

`all` `any`

`map`

`reduce`



built-in

`functools` module

AGGREGATORS

Copyright © 2014

Aggregators

Functions that iterate through an iterable and return a single value that (usually) takes into account every element of the iterable

`min(iterable)` → minimum value in the iterable

`max(iterable)` → maximum value in the iterable

`sum(iterable)` → sum of all the values in the iterable

Associated Truth Values

You should already know this, but let's review briefly:

Every object in Python has an associated **truth value**

`bool(obj)` → **True / False**

Every object has a **True** truth value, except:

- `None`
- `False`
- `0` in any numeric type (e.g. `0`, `0.0`, `0+0j`, ...)
- empty sequences (e.g. `list`, `tuple`, `string`, ...)
- empty mapping types (e.g. `dictionary`, `set`, ...)
- custom classes that implement a `__bool__` or `__len__` method that returns `False` or `0`

which have a **False** truth value

The `any` and `all` functions

`any(iterable)` → returns `True` if any (one or more) element in `iterable` is truthy
 → `False` otherwise

`all(iterable)` → returns `True` if all the elements in `iterable` are truthy
 → `False` otherwise

Leveraging the `any` and `all` functions

Often, we are not particularly interested in the direct truth value of the elements in our iterables

→ want to know if any, or all, satisfy some condition → if the condition is True

A function that takes a single argument and returns True or False is called a `predicate`

We can make `any` and `all` more useful by first applying a predicate to each element of the iterable

Example

Suppose we have some iterable `l = [1, 2, 3, 4, 100]`
and we want to know if: every element is less than `10`

First define a suitable predicate: `pred = lambda x: x < 10`

Apply this predicate to every element of the iterable:

```
results = [pred(1), pred(2), pred(3), pred(4), pred(100)]  
          → [True,      True,      True,      True,      False]
```

Then we use `all` on these results `all(results)` → `False`

How do we apply that predicate?

The `map` function

`map(fn, iterable)`

→ applies `fn` to every element of `iterable`

A comprehension:

`(fn(item) for item in iterable)`

Or even:

```
new_list = []
for item in iterable:
    new_list.append(fn(item))
```



SLICING ITERABLES

NOT JUST SEQUENCE TYPES!

itertools.islice

We know that we can slice sequence types

`seq[i:j:k]`

`seq[slice(i, j, k)]`

We can also slice general iterables (including iterators of course)

→ `islice(iterable, start, stop, step)`

```
from itertools import islice
l = [1, 2, 3, 4]
result = islice(l, 0, 3)
list(result) → [1, 2, 3]
```

→ `islice` returns a lazy iterator

```
list(result) → [] even though l was a list!
```

SELECTING AND FILTERING

Copyright © 2014

The `filter` function

You should already be familiar with the `filter` function → `filter(predicate, iterable)`

→ returns all elements of `iterable` where `predicate(element)` is `True`

`predicate` can be `None` – in which case it is the `identity` function $f(x) \rightarrow x$

→ in other words, truthy elements only will be retained

→ `filter` returns a `lazy iterator`

We can achieve the same result using generator expressions:

`(item for item in iterable if pred(item))` predicate is not `None`

`(item for item in iterable if item)` or `(item for item in iterable if bool(item))`] predicate is `None`

Example

```
filter(lambda x: x < 4, [1, 10, 2, 10, 3, 10]) → 1, 2, 3
```

```
filter(None, [0, '', 'hello', 100, False]) → 'hello', 100
```

→ remember that `filter` returns a (lazy) `iterator`

`itertools.filterfalse`

This works the same way as the `filter` function

but instead of retaining elements where the predicate evaluates to `True`

it retains elements where the predicate evaluates to `False`

Example

```
filterfalse(lambda x: x < 4, [1, 10, 2, 10, 3, 10]) → 10, 10, 10
```

```
filterfalse(None, [0, '', 'hello', 100, False]) → 0, '', False
```

→ `filterfalse` returns a (lazy) `iterator`

itertools.compress

No, this is not a compressor in the sense of say a zip archive!

It is basically a way of **filtering** one iterable, using the truthiness of items in another iterable

```
data = ['a', 'b', 'c', 'd', 'e']
       ↑   ↑   ↑   ↑   ↑
selectors = [True, False, 1, 0] None
```

```
compress(data, selectors) → a, c
```

→ **compress** returns a (lazy) iterator

`itertools.takewhile`

```
takewhile(pred, iterable)
```

The `takewhile` function returns an iterator that will yield items while `pred(item)` is Truthy

→ at that point the iterator is exhausted

even if there are more items in the iterable whose predicate would be truthy

```
takewhile(lambda x: x < 5, [1, 3, 5, 2, 1])      → 1, 3
```

→ `takewhile` returns a (lazy) iterator

`itertools.dropwhile`

```
dropwhile(pred, iterable)
```

The `dropwhile` function returns an iterator that will start iterating (and yield all remaining elements) once `pred(item)` becomes Falsy

```
dropwhile(lambda x: x < 5, [1, 3, 5, 2, 1]) → 5, 2, 1
```

→ `dropwhile` returns a (lazy) iterator

INFINITE ITERATORS

Copyright © 2014

`itertools.count` → lazy iterator

The `count` function is an infinite iterator

similar to `range` → `start, step`

different from `range` → no `stop` → infinite

→ `start` and `step` can be any numeric type

Example

`count(10, 2)` → 10, 12, 14, ...

float
complex
Decimal

`count(10.5, 0.1)` → 10.5, 10.6, 10.7, ...

`takewhile(lambda x: x < 10.8, count(10.5, 0.1))`

→ 10.5, 10.6, 10.7

`itertools.cycle`

→ lazy iterator

The `cycle` function allows us to loop over a finite iterable indefinitely

Example

`cycle(['a', 'b', 'c'])` → 'a', 'b', 'c', 'a', 'b', 'c', ...

Important

If the argument of `cycle` is itself an iterator → iterators becomes exhausted

`cycle` **will still** produce an infinite sequence

→ does not stop after the iterator becomes exhausted

`itertools.repeat` → lazy iterator

The `repeat` function simply yields the same value indefinitely

`repeat('spam')` → 'spam', 'spam', 'spam', 'spam', ...

Optionally, you can specify a count to make the iterator finite

`repeat('spam', 3)` → 'spam', 'spam', 'spam'

Caveat

The items yielded by `repeat` are the same object

→ they each reference the `same` object in memory

CHAINING AND TEEING

Copyright © 2014

Chaining Iterables `itertools.chain(*args)` → lazy iterator

This is analogous to sequence concatenation

but not the same!

- dealing with iterables (including iterators)
- chaining is itself a lazy iterator

We can manually chain iterables this way:

```
iter1  iter2  iter3  
for it in (iter1, iter2, iter3):  
    yield from it
```

Or, we can use `chain` as follows:

```
for item in chain(iter1, iter2, iter3):  
    print(item)
```

Variable number of positional arguments – each argument must be an iterable

Chaining Iterables

What happens if we want to chain from iterables contained inside another, single, iterable?

```
l = [iter1, iter2, iter3]
```

```
chain(l) → l
```

What we really want is to chain `iter1`, `iter2` and `iter3`

We can try this using unpacking: `chain(*l)`

→ produces chained elements from `iter1`, `iter2` and `iter3`

BUT unpacking is **eager** – not lazy!

If `l` was a lazy iterator, we essentially iterated through `l` (not the sub iterators), just to unpack!

This could be a problem if we really wanted the entire chaining process to be lazy

Chaining Iterables

`itertools.chain.from_iterable(it)`

→ lazy iterator

We could try this approach:

```
def chain_lazy(it):
    for sub_it in it:
        yield from sub_it
```

Or we can use `chain.from_iterable`

`chain.from_iterable(it)`

This achieves the same result

→ iterates lazily over `it`

→ in turn, iterates lazily over each iterable in `it`

"Copying" Iterators

`itertools.tee(iterator, n)`

Sometimes we need to iterate through the same iterator multiple times, or even in parallel

We could create the iterator multiple times manually

```
iters = []
for _ in range(10):
    iters.append(create_iterator())
```

Or we can use `tee` in `itertools`

→ returns independent iterators in a tuple

`tee(iterator, 10) → (iter1, iter2, ..., iter10)`



all different objects

Teeing Iterables

One important thing to note

The elements of the returned tuple are **lazy iterators**

- always!
- even if the original argument was not

```
l = [1, 2, 3, 4]
```

```
tee(l, 3) → (iter1, iter2, iter3)
```



all lazy iterators

not lists!

MAPPING AND ACCUMULATION

Mapping and Accumulation

Mapping → applying a callable to each element of an iterable

→ `map(fn, iterable)`

Accumulation → reducing an iterable down to a single value

→ `sum(iterable)` calculates the sum of every element in an iterable

→ `min(iterable)` returns the minimal element of the iterable

→ `max(iterable)` returns the maximal element of the iterable

→ `reduce(fn, iterable, [initializer])`

→ `fn` is a function of two arguments

→ applies `fn` cumulatively to elements of iterable

map

You should already be familiar with `map` → quick review

`map(fn, iterable)` applies `fn` to every element of `iterable`, and returns an iterator (lazy)

→ `fn` must be a callable that requires a single argument

```
map(lambda x: x**2, [1, 2, 3, 4]) → 1, 4, 9, 16 → lazy iterator
```

Of course, we can easily do the same thing using a generator expression too

```
maps = (fn(item) for item in iterable)
```

reduce

You should already be familiar with `reduce` → quick review

Suppose we want to find the sum of all elements in an iterable: `l = [1, 2, 3, 4]`

`sum(l)` → $1 + 2 + 3 + 4 = 10$

`reduce(lambda x, y: x + y, l)` →
→ 1
→ $1 + 2 = 3$
→ $3 + 3 = 6$
→ $6 + 4 = 10$

To find the product of all elements:

`reduce(lambda x, y: x * y, l)` →
→ 1
→ $1 * 2 = 2$
→ $2 * 3 = 6$
→ $6 * 4 = 24$

We can specify a different "start" value in the reduction

`reduce(lambda x, y: x + y, l, 100)` → 110

`itertools.starmap`

`starmap` is very similar to `map`

- it unpacks every sub element of the iterable argument, and passes that to the map function
- useful for mapping a multi-argument function on an iterable of iterables

```
l = [ [1, 2], [3, 4] ]           map(lambda item: item[0] * item[1], l) → 2, 12
```

We can use `starmap`: `starmap(operator.mul, l)` → 2, 12

we could also just use a generator expression to do the same thing:

```
(operator.mul(*item) for item in l)
```

We can of course use iterables that contain more than just two values:

```
l = [ [1, 2, 3], [10, 20, 30], [100, 200, 300] ]  
starmap(lambda: x, y, z: x + y + z, l) → 6, 60, 600
```

`itertools.accumulate(iterable, fn)` → lazy iterator

The `accumulate` function is very similar to the `reduce` function

But it returns a (lazy) iterator producing all the intermediate results

→ `reduce` only returns the final result

Unlike `reduce`, It does not accept an initializer

Note the argument order is not the same!

`reduce(fn, iterable)`
`accumulate(iterable, fn)`

→ in `accumulate`, `fn` is optional

→ defaults to addition

ZIPPING ITERABLES

Copyright © 2018

DATA SCIENCE
INTERVIEW
QUESTIONS

The `zip` Function

→ lazy iterator

We have already seen the `zip` function

It takes a variable number of positional arguments – each of which are iterables

It returns an iterator that produces tuples containing the elements of the iterables, iterated one at a time

It stops immediately once one of the iterables has been completely iterated over

→ zips based on the shortest iterable

```
zip([1, 2, 3], [10, 20], ['a', 'b', 'c', 'd'])
```

→ `(1, 10, 'a'), (2, 20, 'b')`

```
itertools.zip_longest(*args, [fillvalue=None])
```

Sometimes we want to zip, but based on the longest iterable

→ need to provide a **default** value for the "holes" → **fillvalue**

```
zip([1, 2, 3], [10, 20], ['a', 'b', 'c', 'd'])  
→ (1, 10, 'a'), (2, 20, 'b')
```

```
zip_longest([1, 2, 3], [10, 20], ['a', 'b', 'c', 'd'])  
→ (1, 10, 'a'), (2, 20, 'b'), (3, None, 'c'), (None, None, 'd')
```

```
zip_longest([1, 2, 3], [10, 20], ['a', 'b', 'c', 'd'], -1)  
→ (1, 10, 'a'), (2, 20, 'b'), (3, -1, 'c'), (-1, -1, 'd')
```

GROUPING

Copyright © 2014

Grouping

Sometimes we want to loop over an iterable of elements

but we want to **group** those elements as we iterate through them

Suppose we have an iterable containing tuples, and we want to group based on the first element of each tuple

(1, 10, 100)	group 1
(1, 11, 101)	
(1, 12, 102)	
(2, 20, 200)	group 2
(2, 21, 201)	
(3, 30, 300)	
(3, 31, 301)	group 3
(3, 32, 302)	

We would like to iterate using this kind of approach:

key → 1
(1, 10, 100)
(1, 11, 101)
(1, 12, 102)

key → 2
(2, 20, 200)
(2, 21, 201)

key → 3
(3, 30, 300)
(3, 31, 301)
(3, 32, 302)

```
for key, group in groups:  
    print(key)  
    for item in group:  
        print(item)
```

`itertools.groupby(data, [keyfunc])` → lazy iterator

The `groupby` function allows us to do precisely that

→ normally specify `keyfunc` which calculates the `key` we want to use for grouping

`iterable`

(1, 10, 100) Here we want to group based on the 1st element of each tuple

(1, 11, 101) → grouping key `lambda x: x[0]`

(1, 12, 102)

(2, 20, 200) `groupby(iterable, lambda x: x[0])`

(2, 21, 201) → iterator → of tuples (`key, sub_iterator`)

(3, 30, 300) 1, `sub_iterator` → (1, 10, 100), (1, 11, 101), (1, 12, 102)

(3, 31, 301) 2, `sub_iterator` → (2, 20, 200), (2, 21, 201)

(3, 32, 302) 3, `sub_iterator` → (3, 30, 300), (3, 31, 301), (3, 32, 302)

note how the sequence is sorted by the grouping key!

Important Note

The sequence of elements produced from the "sub-iterators" are all produced from the **same** underlying iterator

```
iterable  
(1, 10, 100)  
(1, 11, 101)  
(1, 12, 102)  
(2, 20, 200)  
(2, 21, 201)  
(3, 30, 300)  
(3, 31, 301)  
(3, 32, 302)
```

```
groups = groupby(iterable, lambda x: x[0])  
next(groups) 1, sub_iterator → (1, 10, 100), (1, 11, 101), (1, 12, 102)  
next(groups) 2, sub_iterator → (2, 20, 200), (2, 21, 201)  
next(groups) 3, sub_iterator → (3, 30, 300), (3, 31, 301), (3, 32, 302)
```

The diagram illustrates the state of the iterable and the groups object after each call to `next(groups)`. It shows two parallel sequences of elements. The left sequence represents the original iterable, and the right sequence represents the groups object. Arrows indicate which elements of the iterable become the first group, the second group, and so on.

- Initial state:
 - Iterable: (1, 10, 100), (1, 11, 101), (1, 12, 102), (2, 20, 200), (2, 21, 201), (3, 30, 300), (3, 31, 301), (3, 32, 302)
 - Groups: None
- After `next(groups)`:
 - Iterable: (1, 10, 100), (1, 11, 101), (1, 12, 102), (2, 20, 200), (2, 21, 201), (3, 30, 300), (3, 31, 301), (3, 32, 302)
 - Groups: 1, sub_iterator → (1, 10, 100), (1, 11, 101), (1, 12, 102)
- After `next(groups)`:
 - Iterable: (2, 20, 200), (2, 21, 201), (3, 30, 300), (3, 31, 301), (3, 32, 302)
 - Groups: 2, sub_iterator → (2, 20, 200), (2, 21, 201)
- After `next(groups)`:
 - Iterable: (3, 30, 300), (3, 31, 301), (3, 32, 302)
 - Groups: 3, sub_iterator → (3, 30, 300), (3, 31, 301), (3, 32, 302)

`next(groups)` actually **iterates** through all the elements of the current "sub-iterator" before proceeding to the **next** group

COMBINATORICS

Copyright © 2014 Pearson Education, Inc.

The `itertool` module contains a few functions for generating

permutations

combinations

It also has a function to generate the Cartesian product of multiple iterables

All these functions return `lazy` iterators

Cartesian Product

$$\{1, 2, 3\} \times \{a, b, c\}$$

$$(1, a)$$

$$(2, a)$$

$$(3, a)$$

$$(1, b)$$

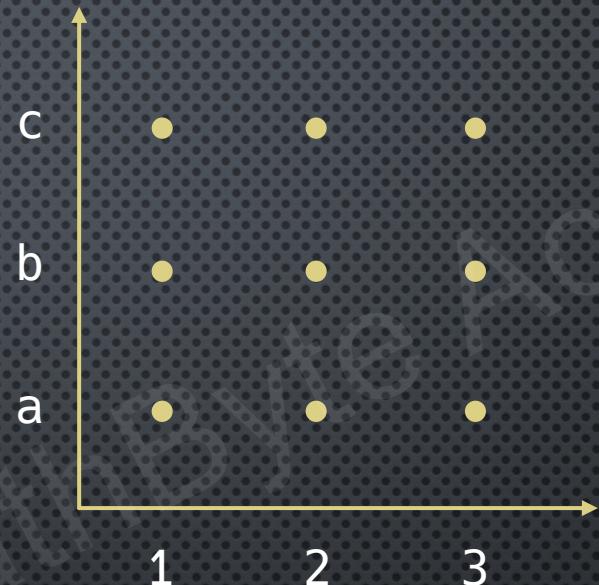
$$(2, b)$$

$$(3, b)$$

$$(1, c)$$

$$(2, c)$$

$$(3, c)$$



2-dimensional: $A \times B = \{(a, b) | a \in A, b \in B\}$

n-dimensional: $A_1 \times \cdots \times A_n = \{(a_1, a_2, \dots, a_n) | a_1 \in A_1, \dots, a_n \in A_n\}$

Cartesian Product

Let's say we wanted to generate the Cartesian product of two lists:

```
l1 = [1, 2, 3]      l2 = ['a', 'b', 'c', 'd']      → notice not same length
```

```
def cartesian_product(l1, l2):
    for x in l1:
        for y in l2:
            yield (x, y)
```

```
cartesian_product(l1, l2)
```

```
→ (1, 'a'), (1, 'b'), (1, 'c'), (1, 'd'), ..., (3, 'd')
```

`itertools.product(*args)` → lazy iterator

`l1 = [1, 2, 3]` `l2 = ['a', 'b', 'c', 'd']`

`product(l1, l2)` → `(1, 'a'), (1, 'b'), (1, 'c'), (1, 'd'), ..., (3, 'd')`

`l3 = [100, 200]`

`product(l1, l2, l3)` → `(1, 'a', 100), (1, 'a', 200),
(1, 'b', 100), (1, 'b', 200),
(1, 'c', 100), (1, 'c', 200),
...
(3, 'd', 100), (3, 'd', 200)`

Permutations

This function will produce all the possible permutations of a given iterable

In addition, we can specify the length of each permutation

→ maxes out at the length of the iterable

`itertools.permutations(iterable, r=None)`

→ `r` is the size of the permutation

→ `r = None` means length of each permutation is the length of the iterable

Elements of the iterable are considered **unique** based on their **position**, not their value

→ if iterable produces repeat values
then permutations will have repeat values too

Combinations

Unlike permutations, the order of elements in a combination is not considered

→ OK to always sort the elements of a combination

Combinations of length **r**, can be picked from a set

- without replacement
 - once an element has been picked from the set it **cannot** be picked again
- with replacement
 - once an element has been picked from the set it **can** be picked again

```
itertools.combinations(iterable, r)
```

```
itertools.combinations_with_replacement(iterable, r)
```

Just like for permutations:

the elements of an iterable are **unique** based on their **position**, not their value

The different combinations produced by these functions are **sorted** based on the **original ordering** in the **iterable**

PROJECT

Copyright © 2014

Data Files

You are given four data files

`personal_info.csv`

`vehicles.csv`

`employment.csv`

`update_status.csv`

Each file contains a common key that uniquely identifies each row – `SSN`

You are guaranteed that every SSN number

- appears only once in every file
- is present in all 4 files
- the order of SSN in each file is the same

To make the approach easier, I am going to break it down into multiple smaller goals

Goal 1

Create (lazy) iterators for each of the four files

- returns named tuples
- data types are appropriate (string, date, int, etc)
- the 4 iterators are independent of each other (for now)

You will want to make use of the standard library module `csv` for this

Reading CSV Files

CSV files are files that contain multiple lines of data → strings

The individual data fields in a row are:

delimited by some separating character → comma, tab are common

in addition, individual fields may be wrapped in further delimiters → quotes are common

→ this allows the field value to contain what may be otherwise interpreted as a delimiter

Example

1,hello,world → 3 values: 1 hello world

1, "hello,world" → 2 values: 1 hello, world

Reading CSV Files

1,hello,world

1,"hello, world"

Simply splitting on the comma is not going to work in the second example! → 1 "hello world"

`csv.reader` is exactly what we need → lazy iterator
→ we can tell it what the delimiter is
→ we can tell it what the quote character is

Example

Mueller-Rath,Human Resources,05-8069298,123-88-3381
"Schumm, Schumm and Reichert",Engineering,73-3839744,125-07-9434

```
def read_file(file_name):
    with open(file_name) as f:
        reader = csv.reader(f, delimiter=',', quotechar='''')
        yield from reader
```

→ yields lists of strings containing each field value

Goal 2

Create a single iterable that combines all the data from all four files

- try to re-use the iterators you created in Goal 1
- by combining I mean one row per SSN containing data from all four files in a single named tuple

Once again, make sure returned data is a single named tuple containing all fields

When you "combine" the data, make sure the SSN's match!

Remember that all the files are already sorted by SSN, and that each SSN appears once, and only once, in every file

- viewing files side by side, all the row SSN's will align correctly

Don't repeat the SSN 4 times in the named tuple – once is enough!

Goal 3

Some records are considered **stale** (not updated recently enough)

A record is considered stale if the `last update date < 3/1/2017`

The update date is located in the `update_status.csv` file

Modify your iterator from Goal 2 to filter out stale records

Make sure your iterator remains **lazy**!

Goal 4

For non-stale records, generate lists of number of car makes by gender

If you do this correctly, the largest groups for each gender are:

Female → Ford and Chevrolet (both have 42 persons in those groups)

Male → Ford (40 persons in the group)

Good luck!

CONTEXT MANAGERS

Copyright © 2018

What is a context?

Oxford dictionary: The circumstances that form the setting for an event, statement, or idea, and in terms of which it can be fully understood.

In Python: the state surrounding a section of code

```
# module.py  
f = open('test.txt', 'r')  
print(f.readlines())  
f.close()
```

global scope

f → a file object

when `print(f.readlines())` runs, it has a context in which it runs

→ global scope

Managing the context of a block of code

Consider the open file example:

```
# module.py  
  
f = open('test.txt', 'r')  
perform_work(f)  
f.close()
```

There could be an exception before we close the file → file remains open!

Need to better "manage" the context that `perform_work(f)` needs

```
f = open('test.txt', 'r')  
  
try:  
    perform_work(f)  
finally:  
    f.close()
```

this works

→ writing `try/finally` every time can get cumbersome
→ too easy to forget to close the file

Context Managers

- create a context (a minimal amount of state needed for a block of code)
 - execute some code that uses variables from the context
- automatically clean up the context when we are done with it

→ enter context

→ open file

→ work within context

→ read the file

→ exit context

→ close the file

Example

```
with open('test.txt', 'r') as f: ← create the context → open file
```

```
    print(f.readlines()) ← work inside the context
```

```
← exit the context → close file
```

Context managers manage data in our scope → on entry
→ on exit

Very useful for anything that needs to provide Enter / Exit Start / Stop Set / Reset

- open / close file
- start db transaction / commit or abort transaction
- set decimal precision to 3 / reset back to original precision

CONTEXT MANAGERS

Copyright © 2018

try...finally...

The **finally** section of a **try** always executes

```
try:  
    ...  
except:  
    ...  
finally:  
    ...
```

always executes
even if an exception occurs in **except** block

Works even if inside a function and a **return** is in the **try** or **except** blocks

Very useful for writing code that should execute no matter what happens

But this can get cumbersome!

There has to be a better way!

Pattern

create some object

do some work with that object

clean up the object after we're done using it

We want to make this easy

→ automatic cleanup after we are done using the object

Context Managers

PEP 343

object returned from context (optional)

```
with context as obj_name:  
    # with block (can use obj_name)
```

after the with block, context is cleaned up automatically

Example

```
with open(file_name) as f:      ← enter the context      (optional) an object is returned  
    # file is now open  
    # file is now closed      ← exit the context
```

The context management protocol

Classes implement the **context management protocol** by implementing two methods:

`__enter__` setup, and optionally return some object

`__exit__` tear down / cleanup

```
with CtxManager() as obj:  
    # do something  
# done with context
```

```
mgr = CtxManager()  
obj = mgr.__enter__()  
try:  
    # do something  
finally:  
    # done with context  
    mgr.__exit__()
```

over simplified
exception handling

Use Cases

Very common usage is for opening a file (creating resource) and closing the file (releasing resource)

Context managers can be used for much more than creating and releasing resources

Common Patterns

- Open – Close
- Lock – Release
- Change – Reset
- Start – Stop
- Enter – Exit

Examples

- file context managers
- Decimal contexts

How Context Protocol Works

works in conjunction with a `with` statement

```
my_obj = MyClass()
```

works as a regular class

`__enter__`, `__exit__` were not called

```
class MyClass:  
    def __init__(self):  
        # init class  
  
    def __enter__(self):  
        return obj  
  
    def __exit__(self, + ...):  
        # clean up obj
```

```
with MyClass() as obj:
```

→ creates an instance of `MyClass` → no associated symbol, but an instance exists

→ calls `my_instance.__enter__()` → `my_instance`

→ return value from `__enter__` is assigned to `obj`

(not the instance of `MyClass` that was created)

after the `with` block, or if an exception occurs inside the `with` block:

→ `my_instance.__exit__` is called

Scope of `with` block

The `with` block is not like a function or a comprehension

The scope of anything in the `with` block (including the object returned from `__enter__`)
is in the same scope as the `with` statement itself

```
# module.py
with open(fname) as f:
    row = next(f)
print(f)
print(row)
```

`f` is a symbol in global scope

`row` is also in the global scope

`f` is closed, but the symbol exists

`row` is available and has a value

The `__enter__` Method

```
def __enter__(self):
```

This method should perform whatever setup it needs to

It can optionally return an object → as `returned_obj`

That's all there is to this method

The `__exit__` Method

More complicated...

Remember the `finally` in a `try` statement? → always runs even if an exception occurs

`__exit__` is similar → runs even if an exception occurs in `with` block

But should it handle things differently if an exception occurred?

→ maybe → so it needs to know about any exceptions that occurred

→ it also needs to tell Python whether to silence the exception, or let it propagate

The `__exit__` Method

```
with MyContext() as obj:  
    raise ValueError  
  
    print ('done')
```

Scenario 1

`__exit__` receives error, performs some clean up and silences error

`print` statement runs

no exception is seen

Scenario 2

`__exit__` receives error, performs some clean up and let's error propagate

`print` statement does not run

the `ValueException` is seen

The `__exit__` Method

- Needs three arguments:
- the exception type that occurred (if any, `None` otherwise)
 - the exception value that occurred (if any, `None` otherwise)
 - the traceback object if an exception occurred (if any, `None` otherwise)
- Returns `True` or `False`:
- `True` = silence any raised exception
 - `False` = do not silence a raised exception

```
def __exit__(self, exc_type, exc_value, exc_trace):  
    # do clean up work here  
    return True # or False
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-14-39a69b57f322> in <module>()  
      1 with MyContext() as obj:  
----> 2     raise ValueError
```

ADDITIONAL USES

Copyright © 2014
National Education Association

Pattern: Open - Close

Open File

 operate on open file

Close File

Open socket

 operate on socket

Close socket

Pattern: Start - Stop

Start database transaction

 perform database operations

Commit or rollback transaction

Start timer

 perform operations

Stop timer

Pattern: Lock - Release

acquire thread lock

perform some operations

release thread lock

Pattern: Change - Reset

change Decimal context precision

 perform some operations using the new precision

reset Decimal context precision back to original value

redirect stdout to a file

 perform some operations that write to stdout

reset stdout to original value

Pattern: Wacky Stuff!

```
with ListMaker(title='Items', prefix='-',  
              indent=3, stdout='myfile.txt') as lm:  
  
    lm.print('Item 1')                                >> myfile.txt  
  
    with lm :  
        lm.print('item 1a')  
        lm.print('item 1b')  
    lm.print(Item 2)  
  
    with lm :  
        lm.print('item 2a')  
        lm.print('item 2b')
```

Items

- Item 1
 - item 1a
 - item 1b
- Item 2
 - item 2a
 - item 2b

GENERATORS AND CONTEXT MANAGERS

Copyright © 2014, Mark Lutz

Context Manager Pattern

create context manager

enter context (and, optionally, receive an object)

do some work

exit context

```
with open(file_name) as f:  
    data = file.readlines()
```

Mimic Pattern using a Generator

```
def open_file(fname, mode):  
    f = open(fname, mode)  
    try:  
        yield f  
    finally:  
        f.close()
```

```
ctx = open_file('file.txt', 'r')  
f = next(ctx)  
  
try:  
    # do work with file  
finally:  
    try:  
        next(ctx)  
    except StopIteration:  
        pass
```

```
ctx = open_file('file.txt', 'r')
```

```
f = next(ctx)      opens file, and yields it
```

```
next(ctx)          closes file
```

→ StopIteration exception

This works in general

```
def gen(args):
    # do set up work here

    try:
        yield object

    finally:
        # clean up object here

        ctx = gen(...)
        obj = next(ctx)

        try:
            # do work with obj
        finally:
            try:
                next(ctx)
            except StopIteration:
                pass
```

This is quite clunky still

but you should see that we can almost
create a context manager pattern using
a generator function!

Creating a Context Manager from a Generator Function

```
def open_file(fname, mode): ← generator function
    f = open(fname, mode)
    try:
        yield f
    finally:
        f.close()

class GenContext:
    def __init__(self, gen):
        self.gen = gen

    def __enter__(self):
        obj = next(self.gen)
        return obj

    def __exit__(self, exc_type, exc_value, exc_tb):
        next(self.gen)
        return False
```

generator object → gen = open_file('test.txt', 'w')

f = next(gen)

do work with f

next(f) → closes f

gen = open_file('test.txt', 'w')

with GenContext(gen) as f:

do work

DECORATING GENERATOR FUNCTIONS

Copyright © 2014



So far...

we saw how to create a context manager using a class and a generator function

```
def gen_function(args):  
    ...  
    try:  
        yield obj ← single yield the return value of __enter__  
    finally:  
        ...
```

```
class GenContextManager:  
    def __init__(gen_func):  
        self.gen = gen_func()
```

```
    def __enter__(self):  
        return next(self.gen) ← returns what was yielded
```

```
    def __exit__(self, ...):  
        next(self.gen) ← runs the finally block
```

Usage

```
with GenContextManager(gen_func):  
    ...
```

We can tweak this a bit to also allow passing in arguments to `gen_func`

And usage now becomes:

```
gen = gen_func(args)  
with GenContextManager(gen):  
    ...
```

This works, but we have to create the generator object first, and use the `GenContextManager` class

→ lose clarity of what the context manager is

```
class GenContextManager:  
    def __init__(gen_obj):  
        self.gen = gen_obj  
  
    def __enter__(self):  
        return next(self.gen)  
  
    def __exit__(self, ...):  
        next(self.gen)
```

Using a decorator to encapsulate these steps

```
gen = gen_func(args)  
with GenContextManager(gen):  
    ...
```

```
def contextmanager_dec(gen_fn):  
    def helper(*args, **kwargs):  
        gen = gen_fn(*args, **kwargs)  
        return GenContextManager(gen)  
    return helper
```

```
class GenContextManager:  
    def __init__(gen_obj):  
        self.gen = gen_obj  
  
    def __enter__(self):  
        return next(self.gen)  
  
    def __exit__(self, ...):  
        next(self.gen)
```

Usage Example

```
@contextmanager_dec  
def open_file(f_name):  
    f = open(f_name)  
    try:  
        yield f  
  
    finally:  
        f.close()
```

→ `open_file = contextmanager_dec(open_file)`

→ `open_file` is now actually the `helper` closure

calling `open_file(f_name)`

→ calls `helper(f_name)` [free variable `gen_fn = open_file`]

→ creates the generator object

→ returns `GenContextManager` instance

→ `with open_file(f_name)`

```
def contextmanager_dec(gen_fn):  
    def helper(*args, **kwargs):  
        gen = gen_fn(*args, **kwargs)  
        return GenContextManager(gen)  
  
    return helper
```

The `contextlib` Module

One of the goals when context managers were introduced to Python

PEP 343

was to ensure generator functions could be used to easily create them

Technique is basically what we came up with

→ more complex

→ exception handling

→ if an exception occurs in `with` block, needs to be propagated back to generator function

`__exit__(self, exc_type, exc_value, exc_tb)`

→ enhanced generators as coroutines

→ later

This is implemented for us in the standard library:

`contextlib.contextmanager`

→ `decorator` which turns a generator function into a context manager

PROJECT 5

Copyright © 2014 McGraw-Hill Education

Project Setup

In this project you are provided two CSV files

`cars.csv`

`personal_info.csv`

→ first row contains the field names

The basic goal will be to create a context manager that only requires the file name

and provides us an iterator we can use to iterate over the data in those files

The iterator should yield named tuples with field names based on the header row in the CSV file

For simplicity, we assume all fields are just strings

Goal 1

For this goal implement the context manager using a context manager class

i.e. a class that implements the context manager protocol

`__enter__` `__exit__`

Make sure your iterator uses lazy evaluation

If you can, try to create a single class that implements **both** the **context manager** protocol and the **iterator** protocol

Goal 2

For this goal, re-implement what you did in Goal 1, but using a generator function instead

You'll have to use the `@contextmanager` from the `contextlib` module

Information you may find useful

File objects implement the iterator protocol:

```
with open(f_name) as f:  
    for row in f:  
        print(row)
```

But file objects **also** support just reading data using the `read` function

we specify how much of the file to `read` (that can span multiple rows)

when we do this a "read head" is maintained → we can reposition this read head → `seek()`

```
with open(f_name) as f:  
    print(f.read(100))  
    print(f.read(100))  
    f.seek(0)
```

→ reads the first 100 characters → read head is now at 100
→ reads the next 100 characters → read head is now at 200
→ moves read head back to beginning of file

Information you may find useful

CSV files can be read using `csv.reader`

But CSV files can be written in different "styles" → dialects

`john,cleese,42` `john;cleese;42` `john|cleese|42` `john\tcleese\t42`

`"john","cleese","42"` `'john';'cleese';'42'`

The csv module has a `Sniffer` class we can use to auto-determine the specific dialect
→ need to provide it a sample of the csv file

```
with open(f_name) as f:  
    sample = f.read(2000)  
    dialect = csv.Sniffer().sniff(sample)
```

```
with open(f_name) as f:  
    reader = csv.reader(f, dialect)
```

Good Luck!

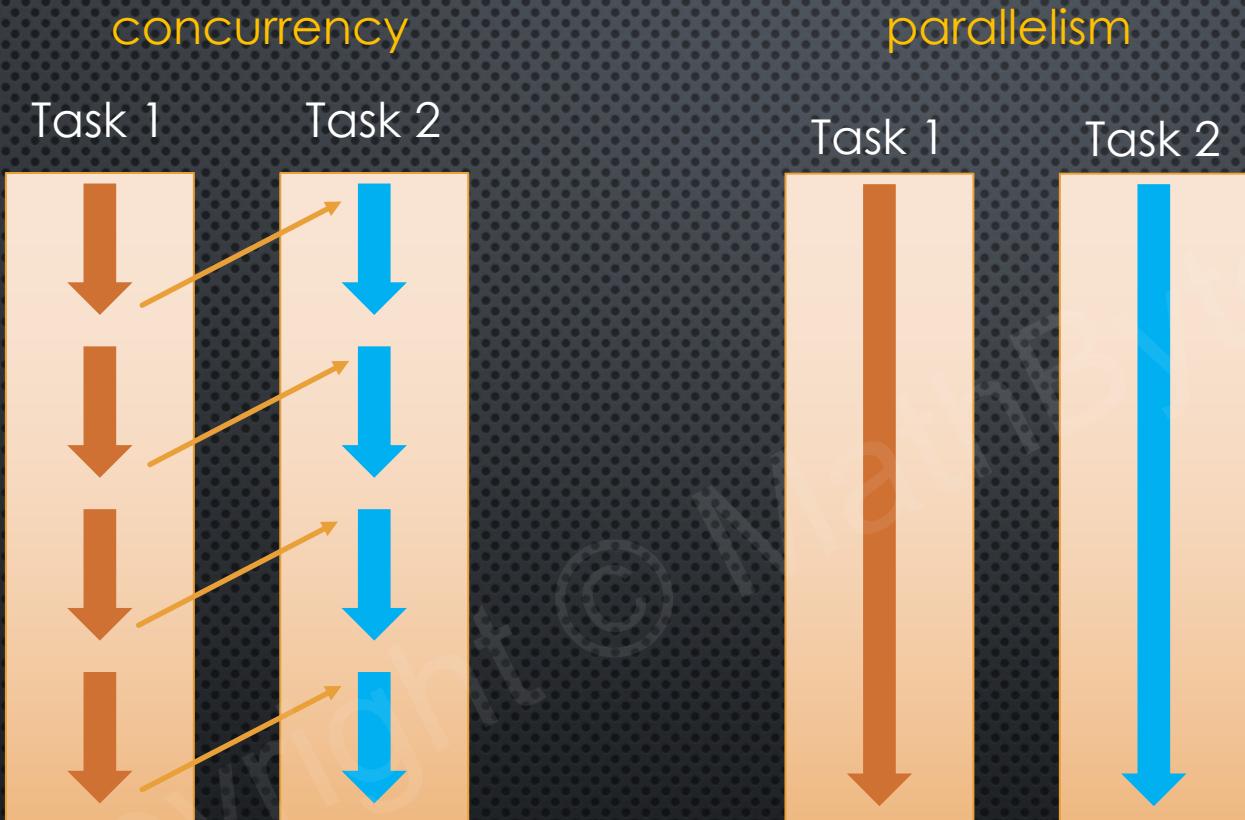
Copyright © 2014

GENERATORS AS COROUTINES

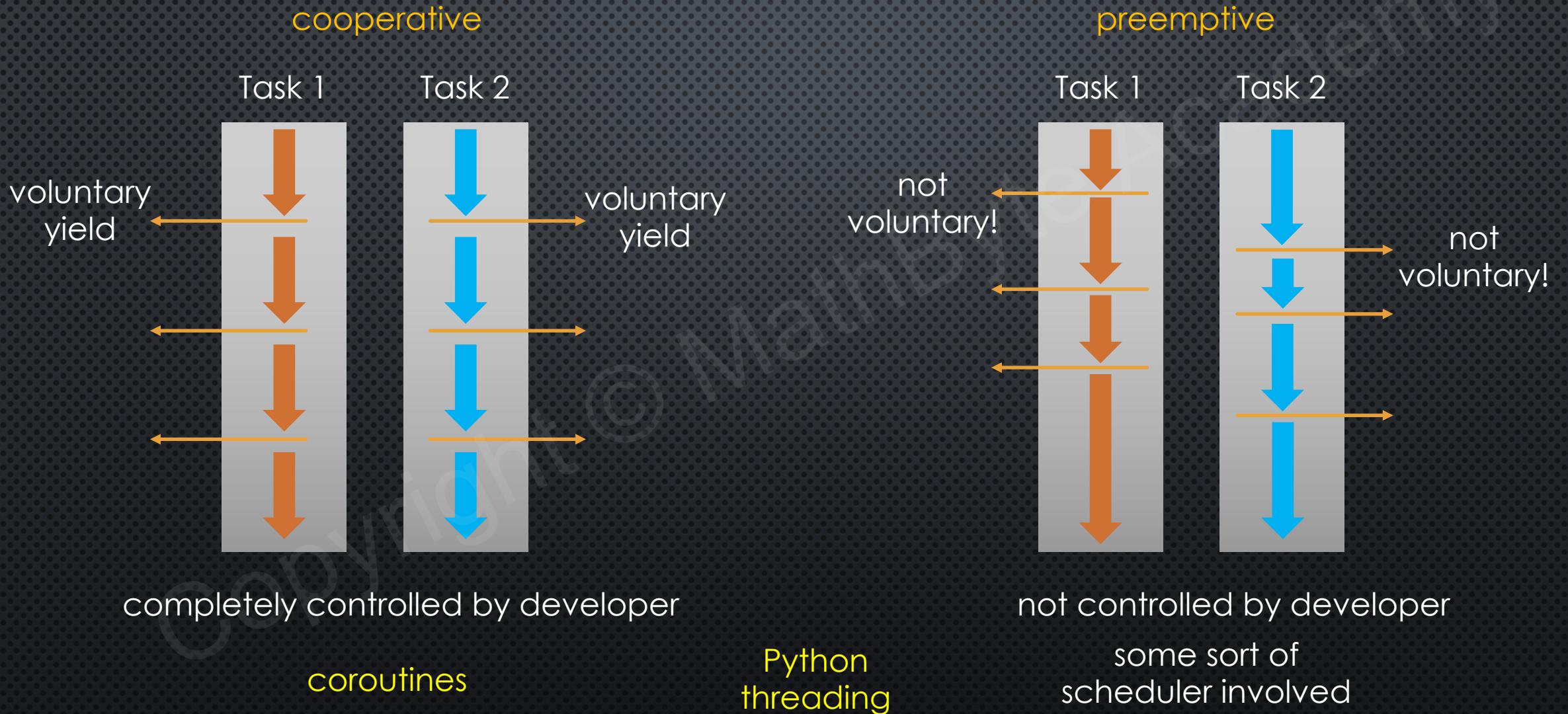
Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Java™ Platform
Micro Edition

Concurrency vs Parallelism



Cooperative vs Preemptive Multitasking



Coroutines

Cooperative multitasking

Concurrent, not parallel

→ Python programs execute on a "single thread"

Global Interpreter Lock → GIL

Two ways to create coroutines in Python

→ generators

→ uses extended form of `yield`

→ recent addition: `asyncio`

→ native coroutines

→ uses `async / await`

This section is **not** about

asyncio

native coroutines

threading

multiprocessing

→ parallelism

This section is **about**

learning the basics of generator-based coroutines

some practical applications of these coroutines

COROUTINES

Copyright © 2018

What is a coroutine?

cooperative routines

subroutines

`running_averages` is in control

```
def running_averages(iterable):  
    avg = averager()  
    for value in iterable:  
        running_average = avg(value)  
        print(running_average)
```

subroutine is called

`inner` is now in control

stack frame
created
(inner)

`running_averages` is back in control

stack frame
destroyed
(inner)

```
def averager():  
    total = 0  
    count = 0  
    def inner(value):  
        nonlocal total  
        nonlocal count  
        total += value  
        count += 1  
        return total / count  
    return inner
```

subroutine terminates

may, or may not
return a value

coroutine

```
def running_averages(iterable):  
    create instance of running_averager  
    start coroutine ——————  
    for value in iterable:  
        send value to running_averager  
        received value back ←  
        print(received value)
```

```
def running_averager():  
    total = 0  
    count = 0  
    running_average = None  
    while True:  
        → wait for value  
        → receive new value  
        calculate new average  
        yield new average ——————
```

stack frame
created
(running_averager)

coroutine is still active
waiting for next value to be sent

We'll come back to this example in another lecture

Abstract Data Structures

What are queues and stacks?

A **queue** is a data structure that supports **first-in first-out** (FIFO) addition/removal of items



A **stack** is a data structure that supports **last-in first-out** addition/removal of items



why **abstract**?

many different ways of creating concrete implementations

Using lists

stack	<code>lst.append(item)</code>	→ appends item to end of list
	<code>lst.pop()</code>	→ removes and returns last element of list
queue	<code>lst.insert(0, item)</code>	→ inserts item to front of list
	<code>lst.pop()</code>	→ removes and returns last element of list

So a `list` can be used for both a stack and a queue

But, inserting elements in a list is quite inefficient!

numbers coming up in a bit...

The deque data structure

Python's `collections` module implements a data structure called `deque`

This is a double-ended queue

→ very efficient at adding / removing items from `both front` and `end` of a collection

```
from collections import deque
```

```
dq = deque()
```

```
dq = deque(iterable) dq = deque(maxlen=n)
```

```
dq.append(item)
```

```
dq.appendleft(item)
```

```
dq.pop()
```

```
dq.popleft()
```

```
dq.clear()
```

```
len(dq)
```

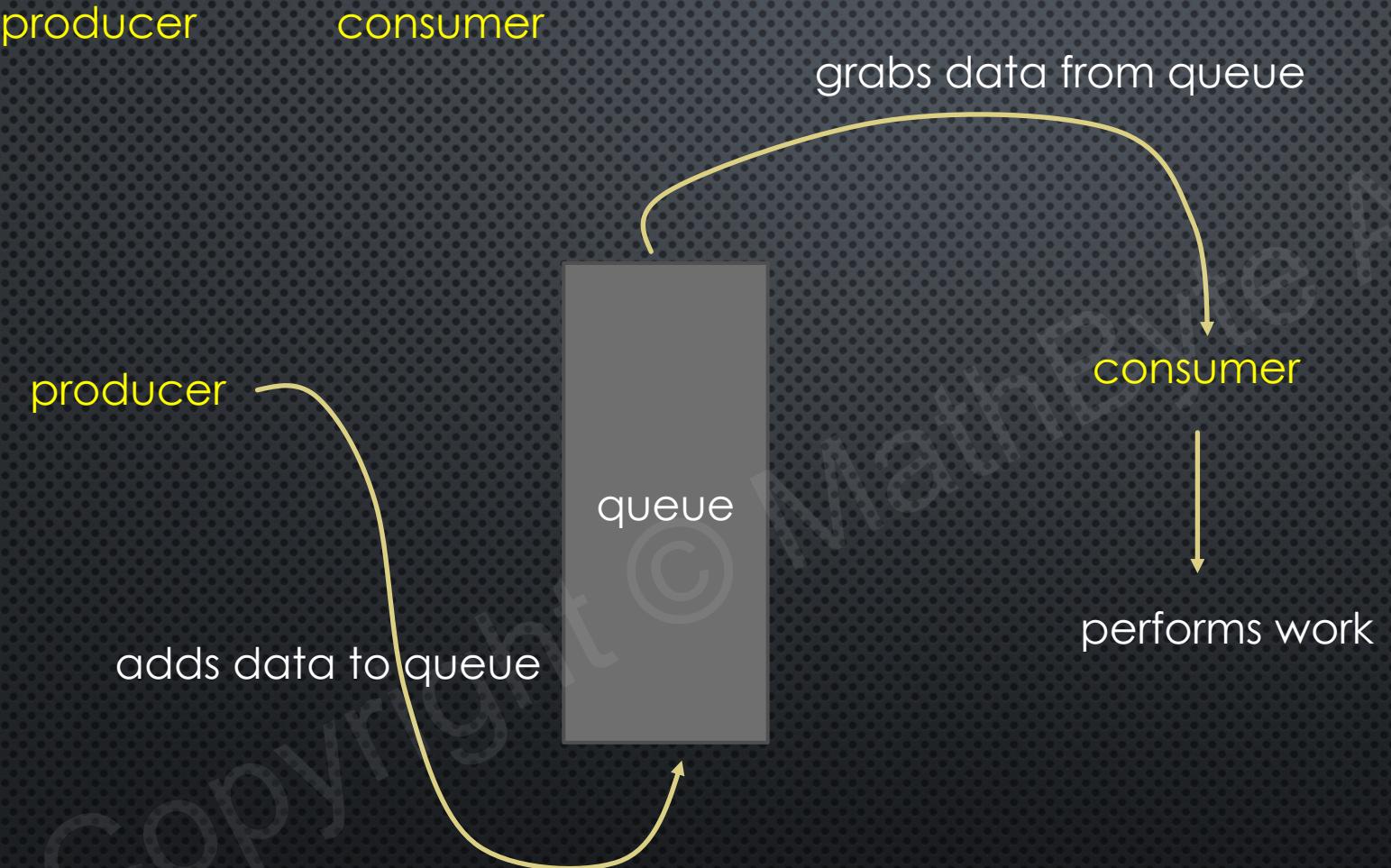
Timings

items = 10_000
tests = 1_000

(times in seconds)

	list	deque	
append (right)	0.87	0.87	--
pop (right)	0.002	0.0005	x4
insert (left)	20.80	0.84	x25
pop (left)	0.012	0.0005	x24

Another use case...



Implementing a Producer/Consumer using Subroutines

- create an "unlimited" deque
- run producer to insert all elements into deque
- run consumer to remove and process all elements in deque

```
def produce_elements(dq):  
    for i in range(1, 100_000):  
        dq.appendleft(i)  
  
def coordinator():  
    dq = deque()  
    producer = produce_elements(dq)  
    consume_elements(dq)
```

```
def consume_elements(dq):  
    while len(dq) > 0:  
        item = dq.pop()  
        print('processing item', item)
```

Implementing a Producer/Consumer using Generators

- create a limited size deque
 - coordinator creates instance of producer generator
 - coordinator creates instance of consumer generator
 - producer runs until deque is filled
 - yields control back to caller
 - consumer runs until deque is empty
 - yields control back to caller
- repeat until producer is "done"
or controller decides to stop

Implementing a Producer/Consumer using Generators

```
def produce_elements(dq, n):
    for i in range(1, n):
        dq.appendleft(i)
        if len(dq) == dq maxlen:
            yield

def coordinator():
    dq = deque(maxlen=10)
    producer = produce_elements(dq, 100_000)
    consumer = consume_elements(dq)
    while True:
        try:
            next(producer)
        except StopIteration:
            break
        finally:
            next(consumer)

def consume_elements(dq):
    while True:
        while len(dq) > 0:
            item = dq.pop()
            # process item
            yield
```

Notice how `yield` is not used to yield values
but to yield control back to controller

GENERATOR STATES

Copyright © 2014 by  McGraw-Hill Education

Generators can be in different states

```
def my_gen(f_name):  
    f = open(f_name)  
    try:  
        for row in f:  
            yield row.split(',')  
    finally:  
        f.close()
```

create the generator

```
rows = my_gen() → CREATED
```

run the generator

```
next(rows) → RUNNING
```

until `yield`

→ SUSPENDED

until generator `return`

→ CLOSED

Inspecting a generator's state

use `inspect.getgeneratorstate` to see the current state of a generator

```
from inspect import getgeneratorstate
```

```
g = my_gen()           getgeneratorstate(g) → GEN_CREATED
```

```
row = next(g)          getgeneratorstate(g) → GEN_SUSPENDED
```

```
lst(g)                getgeneratorstate(g) → GEN_CLOSED
```

(inside the generator code while it is running)

```
getgeneratorstate(g) → GEN_RUNNING
```

SENDING TO GENERATORS

Copyright © 2014 by
National Grid Power Solutions

So far...

We saw how `yield` can produce values

→ use iteration to get the produced values → `next()`

After a value is yielded, the generator is suspended

How about sending data to the generator upon resumption?

Enhancement to generators introduced in Python 2.5

PEP 342

Sending data to a generator

`yield` is actually an expression

it can yield a value (like we have seen before)

```
yield 'hello'
```

it can also `receive` values

it is used just like an expression would

```
received = yield
```

we can combine both

```
received = yield 'hello'
```

- works, but confusing!
- use sparingly



What's happening?

```
def gen_echo():
    while True:
        received = yield
        print('You said:', received)
```

`echo = gen_echo()` → CREATED has not started running yet – not in a suspended state

`next(echo)` → SUSPENDED Python has just yielded (`None`)
generator is suspended at the `yield`

we can resume and send data to the generator at the same time using `send()`

`echo.send('hello')`

generator resumes running exactly at the `yield`

the `yield` expression evaluates to the just received data

then the assignment to `received` is made

What's happening?

```
received = | yield 'python'
```

generator is
suspended here

'python' is yielded and control is returned to caller

caller sends data to generator: `g.send('hello')`

generator resumes

'hello' is the result of the `yield` expression

'hello' is assigned to `received`

generator continues running until the next `yield` or `return`

Priming the generator

```
received = yield 'python'
```

Notice that we can only send data if the generator is **suspended** at a **yield**

So we cannot send data to a generator that is in a **CREATED** state – it must be in a **SUSPENDED** state

```
def gen_echo():
    while True:
        received = yield
        print('You said:', received)
```

echo = gen_echo() → CREATED

echo.send('hello') 

next(echo) → SUSPENDED

echo.send('hello') 

→ yes, a value has been yielded – and we can choose to just ignore it

→ in this example, **None** has been yielded

Priming the generator

Don't forget to **prime** a generator before sending values to it!

- generator must be SUSPENDED to receive data
- always use `next()` to prime

Later we'll see how we can "automatically" prime the generator using a decorator

Using `yield`...

- used for producing data → `yield 'Python'`
- used for receiving data → `a = yield` (technically this produces `None`)

Be careful mixing the two usages in your code

- difficult to understand
- sometimes useful
- often not needed

Example

```
def running_averager():
    total = 0
    count = 0
    running_average = None
    while True:
        value = yield running_average
        total += value
        count += 1
        running_average = total / count
```

```
averager = running_averager()
```

`next(averager)` → primed → `None` has been yielded

`averager.send(10)` → `value` received `10`
→ continues running until next `yield`

→ yields `running_average` → `10`
→ suspended and waiting

`averager.send(30)` → `value` received `30` → eventually yields `20`

CLOSING GENERATORS

Copyright © 2014 NCC Group Ltd.

Consider this generator function...

```
def read_file(f_name):
    f = open(f_name)
    try:
        for row in f:
            yield row
    finally:
        f.close()
```

Suppose the file has 100 rows

→ `yield from f`

```
rows = read_file('test.txt')
for _ in range(10):
    next(rows)
```

- read 10 rows
- file is still `open`
- how do we now `close` the file without iterating through the entire file?

Closing a generator

We have seen the possible generator states created, running, suspended, closed

We can close a generator by calling its `close()` method

```
def read_file(f_name):
    f = open(f_name)
    try:
        for row in f:
            yield row
    finally:
        f.close()

rows = read_file('test.txt')
for _ in range(10):
    next(rows)

rows.close() → finally block runs, and file is closed
```

why did it jump to `finally`? Did an exception occur?

Behind the scenes...

When `.close()` is called, an exception is triggered inside the generator

The exception is a `GeneratorExit` exception

```
def gen():
    try:
        yield 1
        yield 2
    except GeneratorExit:
        print('Generator close called')
    finally:
        print('Cleanup here...')

g = gen()
next(g)
g.close()      → Generator close called
                → Cleanup here...
```

Python's expectations when `close()` is called

- a `GeneratorExit` exception bubbles up
- the generator exits cleanly (returns)
- some other exception is raised from inside the generator
 - the exception is silenced by Python
 - to the caller, everything works "normally"
- exception is seen by caller

if the generator "ignores" the `GeneratorExit` exception and `yields` another value

→ Python raises a `RuntimeError: generator ignored GeneratorExit`

in other words, don't try to catch and ignore a `GeneratorExit` exception

it's perfectly OK not to catch it, and simply let it bubble up

```
def gen():  
    yield 1  
    yield 2
```

```
g = gen()  
next(g)  
g.close()
```



Use in coroutines

Since coroutines are generator functions, it is OK to close a coroutine also

For example, you may have a coroutine that receives data to write to a database

- coroutine opens a transaction when it is primed (`next`)
- coroutine receives data to write to the database
- coroutine commits the transaction when `close()` is called (`GeneratorExit`)
- coroutine aborts (rolls back) transaction if some other exception occurs

SENDING EXCEPTIONS TO GENERATORS

Copyright © 2014

Sending things to coroutines

`.send(data)` → sends `data` to coroutine

`.close()` → sends (throws) a `GeneratorExit` exception to coroutine

we can also "send" any exception to the coroutine → `throwing` an exception to the coroutine

`.throw(exception)`

→ the exception is `raised at the point where the coroutine is suspended`

How `throw()` is handled

→ generator does not catch the exception (does nothing)

 → exception propagates to caller

→ generator catches the exception, and does something

 → yields a value

 → exits (returns)

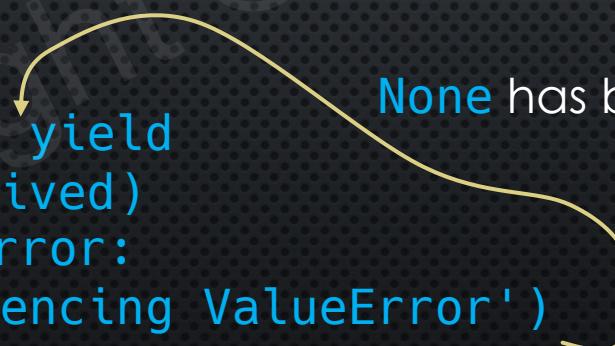
 → raises a different exception

Catch and yield

- generator catches the exception
- handles and silences the exception
- yields a value → generator is now SUSPENDED
- yielded value is the return value of the `.throw()` method

```
def gen():  
    while True:  
        try:  
            received = yield  
            print(received)  
        except ValueError:  
            print('silencing ValueError')
```

None has been yielded



Catch and exit

- generator catches the exception
- generator exits (returns)
- caller receives a **StopIteration** exception → generator is now **CLOSED**
 - this is the same as calling `next()` or `send()` to a generator that returns instead of yielding
 - can think of `throw()` as same thing as `send()`, but causes an **exception** to be sent instead of plain data

```
def gen():  
    while True:  
        try:  
            received = yield  
            print(received)  
        except ValueError:  
            print('silencing ValueError')  
    return None
```

→ **StopIteration**
is seen by caller

Catch and raise different exception

- generator catches the exception
- generator handles exception and raises another exception
- new exception propagates to caller → generator is now **CLOSED**

```
def gen():  
    while True:  
        try:  
            received = yield  
            print(received)  
        except ValueError:  
            print('silencing ValueError')  
            raise CustomException
```

CustomException
is seen by caller

`close()` vs `throw()`

`close()` → `GeneratorExit` exception is raised inside generator

can we just call? `gen.throw(GeneratorExit())`

yes, but...

with `close()`, Python expects the `GeneratorExit`, or `StopIteration` exceptions to propagate, and silences it for the caller

if we use `throw()` instead, the `GeneratorExit` exception is raised inside the caller context (if the generator lets it)

```
try:  
    gen.throw(GeneratorExit())  
except GeneratorExit:  
    pass
```

USING A DECORATOR TO PRIME A COROUTINE

Copyright © 2018, Dr. Christian J. Stach, All Rights Reserved

We always have to prime a coroutine before using it

- very repetitive
- pattern is the same every time

`g = gen()` → creates coroutine instance

`next(g)`
or `g.send(None)` → primes the coroutine

This is a perfect example of using a decorator to do this work for us!

Creating a function to auto prime coroutines

```
def prime( gen_fn ):  
    g = gen_fn( ) —————→ creates the generator  
    next(g) —————→ primes the generator  
    return g —————→ returns the primed generator
```

```
def echo( ):  
    while True:  
        received = yield  
        print(received)
```

```
echo_gen = prime(echo)
```

```
echo_gen.send('hello') → 'hello'
```

A decorator approach

We still have to remember to call the `prime` function for our `echo` coroutine before we can use it

Since `echo` is a coroutine, we know we **always** have to prime it first

So let's write a decorator that will replace our generator function with another function that will automatically prime it when we create an instance of it

```
def coroutine(gen_fn):
    def prime():
        g = gen_fn()
        next(g)
        return g
    return prime
```

```
@coroutine
def echo():
    while True:
        received = yield
        print(received)
```

Understanding how the decorator works

```
def coroutine(gen_fn):  
    def prime():  
        g = gen_fn()  
        next(g)  
        return g  
    return prime
```

```
def echo():  
    while True:  
        received = yield  
        print(received)
```

```
echo = coroutine(echo) [same effect as using @coroutine]
```

→ echo function is now actually the prime function

→ prime is a closure

→ free variable gen_fn is echo

calling echo()

→ calls prime() with gen_fn = echo

```
g = echo()  
next(g)  
return g
```

Expanding the decorator

```
def coroutine(gen_fn):
    def prime():
        g = gen_fn()
        next(g)
        return g
    return prime
```

→ cannot pass arguments to the generator function

```
def coroutine(gen_fn):
    def prime(*args, **kwargs):
        g = gen_fn(*args, **kwargs)
        next(g)
        return g
    return prime
```

YIELD FROM
Two-WAY COMMUNICATIONS

Recall...

```
def subgen():
    for i in range(10):
        yield i
```

We could consume the data from `subgen` in another generator this way:

```
def delegator():
    for value in subgen():
        yield value
```

Instead of using that loop, we saw we could just write:

```
def delegator():
    yield from subgen()
```

With either definition we can call it this way:

```
d = delegator()
next(d)
etc...
```

What is going on exactly?



2-way communications

Can we `send()`, `close()` and `throw()` also?

Yes!

How does the delegator behave when subgenerator returns?

it continues running normally

```
def delegator():
    yield from subgen()
    yield 'subgen closed'
```

```
def subgen():
    yield 1
    yield 2
```

```
d = delegator()
```

```
next(d)      → 1
```

```
next(d)      → 2
```

```
next(d)      → subgen closed
```

```
next(d)      → StopIteration
```

Inspecting the subgenerator

```
from inspect import getgeneratorlocals, getgeneratorstate
```

```
def delegator():
    a = 100
    s = subgen()
    yield from s
    yield 'subgen closed'
```

```
def subgen():
    yield 1
    yield 2
```

```
d = delegator()
```

```
getgeneratorstate(d) → GEN_CREATED
getgeneratorlocals(d) → {}
```

```
next(d) → 1  getgeneratorstate(d) → GEN_SUSPENDED
            getgeneratorlocals(d) → {'a': 100, 's': <gen object>}
```

```
s = getgeneratorlocals(d)['s']  getgeneratorstate(s) → GEN_SUSPENDED
```

```
next(d) → 2          d → SUSPENDED      s → SUSPENDED
```

```
next(d) → 'subgen closed'  d → SUSPENDED      s → CLOSED
```

```
next(d) → StopIteration  d → CLOSED        s → CLOSED
```

YIELD FROM
© SENDING DATA

Copyright © 2014 Microsoft Corporation

`yield from` and `send()`

`yield from` establishes a 2-way communication channel

between **caller** and **subgenerator**

via a delegator → `yield from`



Priming the subgenerator coroutine

We know that before we can `send()` to a coroutine, we have to prime it first → `next(coroutine)`

How does this work with `yield from`?

```
def delegator():
    yield from coro()
```

```
def coro():
    while True:
        received = yield
        print(received)
```

```
d = delegator()
```

before we can send to `d` we have to prime it → `next(d)`

What about `coro()`?

`yield from` will automatically prime the coroutine when necessary

Sending data to the subgenerator

Once the delegator has been primed

data can be sent to it using `send()`

```
def delegator():
    yield from coro()
```

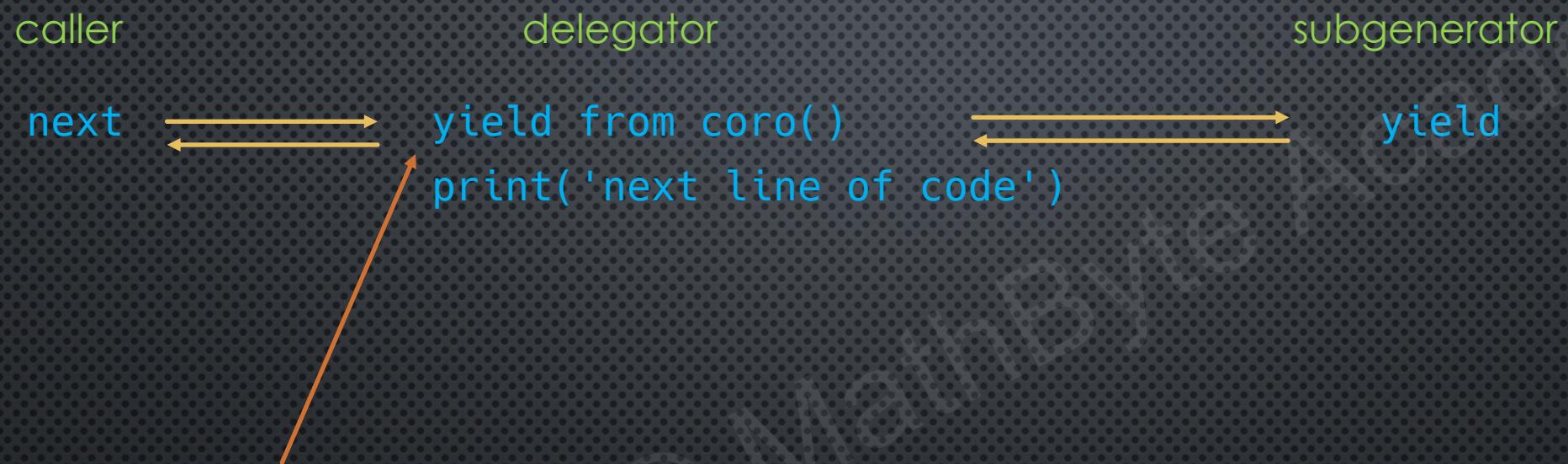
```
d = delegator()
```

```
next(d)
```

```
d.send('python')      → python is printed by coroutine
```

```
def coro():
    while True:
        received = yield
        print(received)
```

Control Flow



delegator is "stuck" here until subgenerator **closes**

then it resumes running the rest of the code

Multiple Delegators → pipeline

```
def coro( ):      def gen1( ):
    ...            yield from gen2( )
    yield           def gen2( ):
    ...            yield from coro()
```

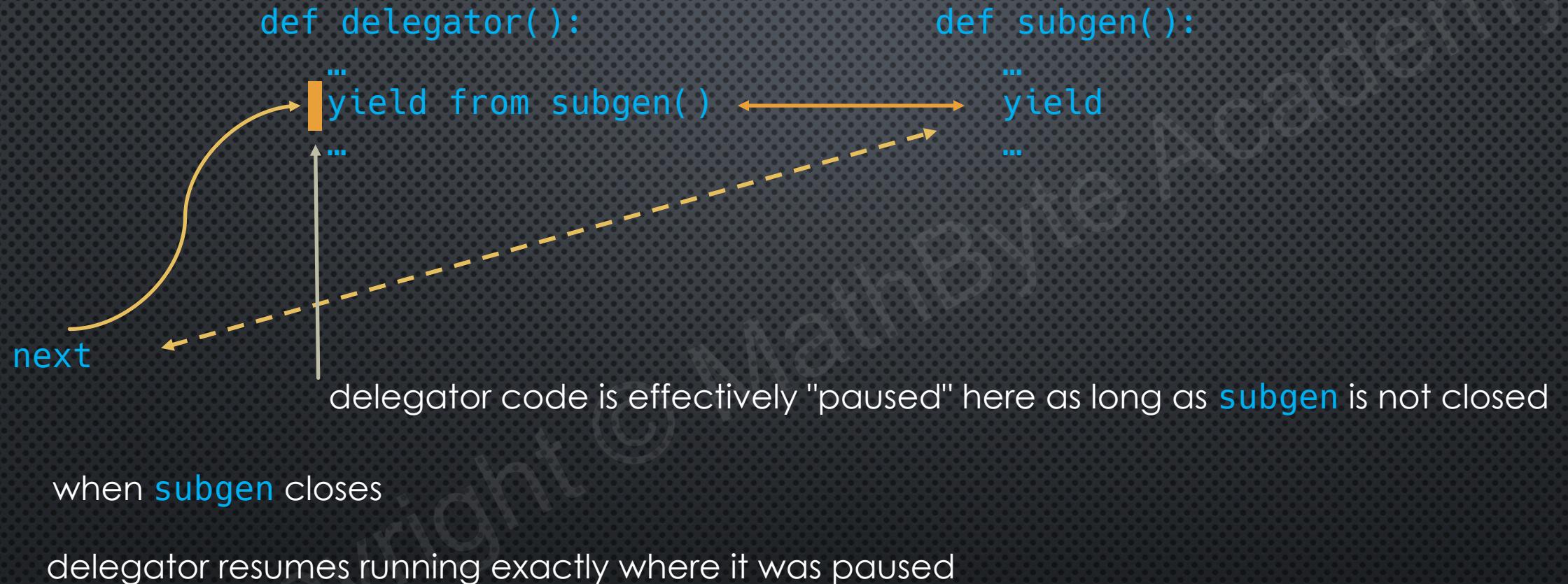
```
d = gen1()
```

```
caller ←→ gen1 ←→ gen2 ←→ coro
```

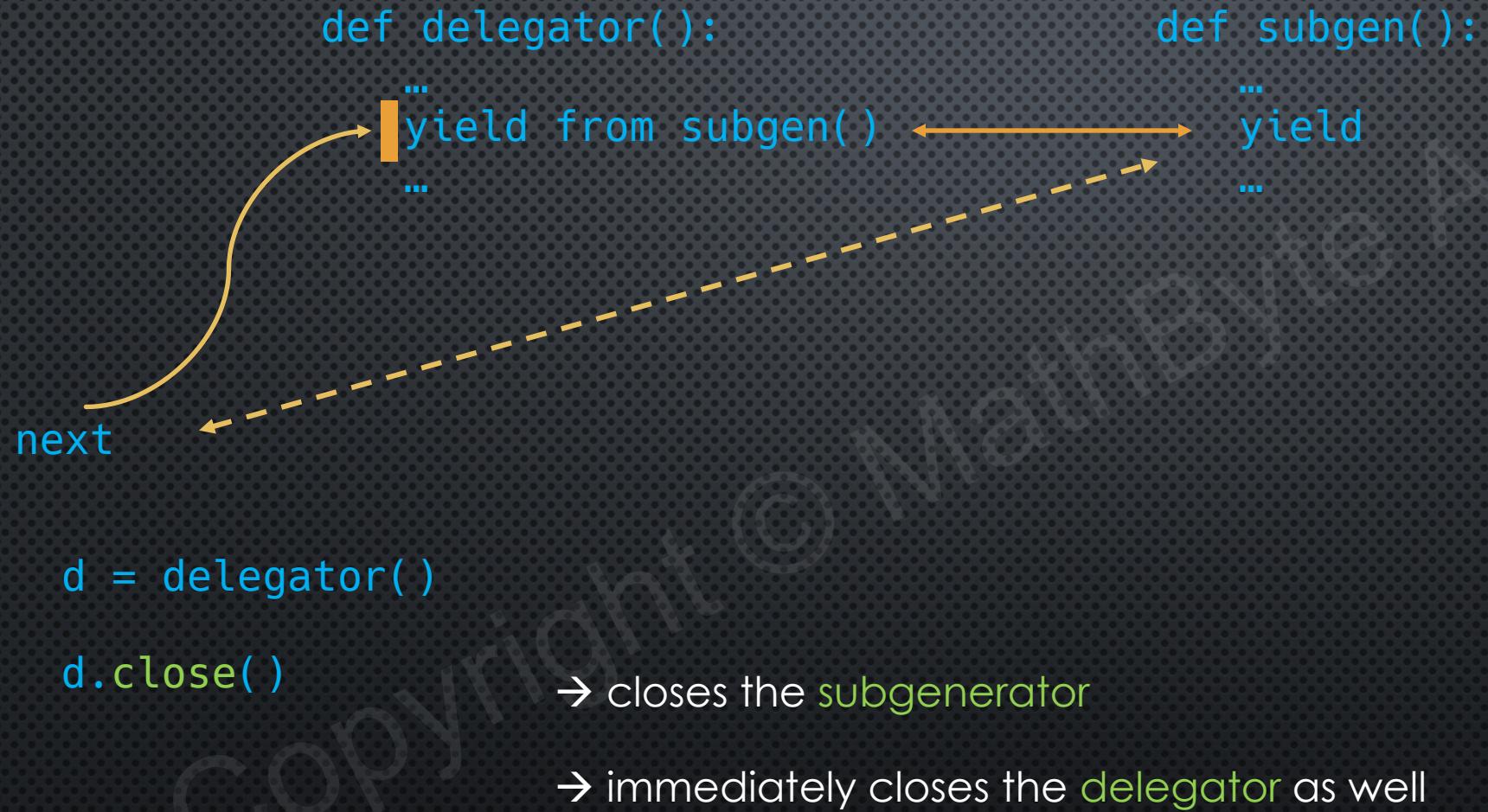
this can even be recursive

YIELD FROM CLOSING AND RETURN

Closing the subgenerator



Closing the delegator



Returning from a generator

A generator can `return`

→ `StopIteration`

The returned value is `embedded` in the `StopIteration` exception

→ we can extract that value

```
try:  
    next(g)  
except StopIteration as ex:  
    print(ex.value)
```

→ so can Python!

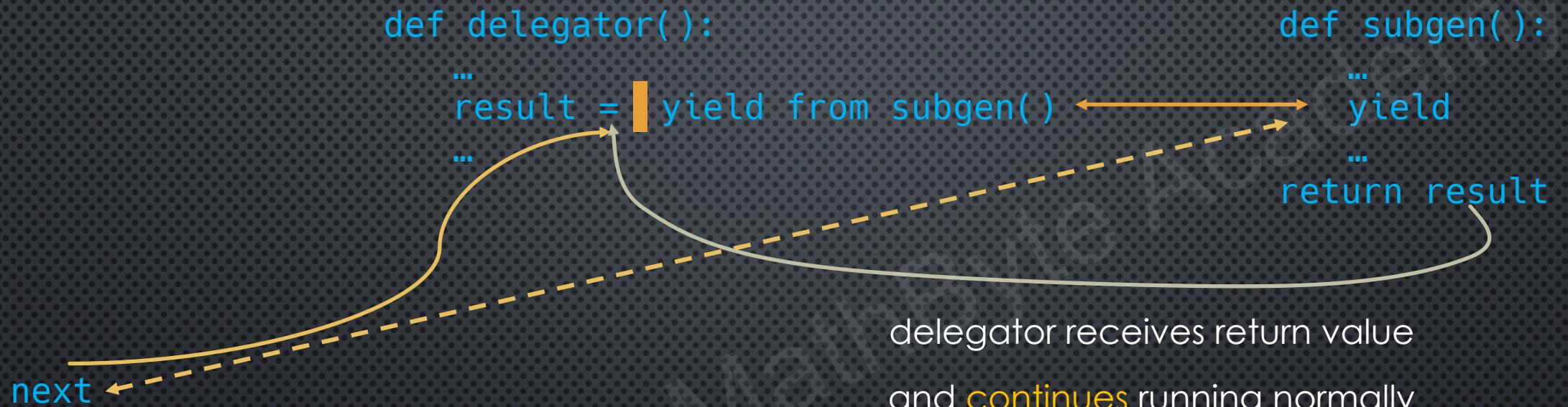
Returning from a subgenerator

`yield from` is an expression

It evaluates to the returned value of the subgenerator

```
result = yield from subgen( )  
def subgen( ):  
    ...  
    yield  
    ...  
    return result
```

Returning from a subgenerator



yield from

→ establishes conduit

subgenerator returns

- conduit is closed
- yield from evaluates to the returned value
- delegator resumes running

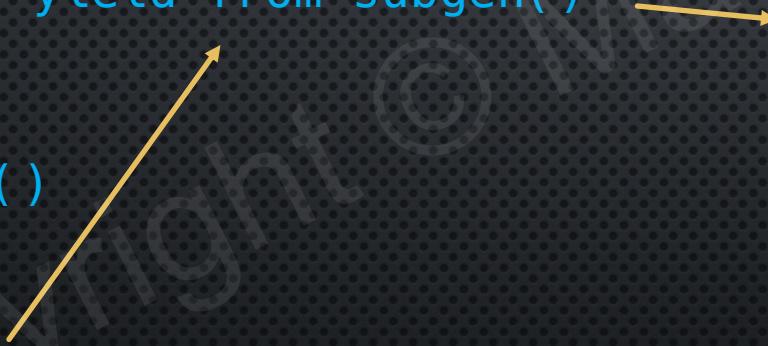
YIELD FROM THROWING EXCEPTIONS

Throwing Exceptions

We can throw exceptions into a generator using the `throw()` method

→ works with delegation as well

```
def delegator():  
    yield from subgen()  
  
d = delegator()  
  
d.throw(Exc)
```

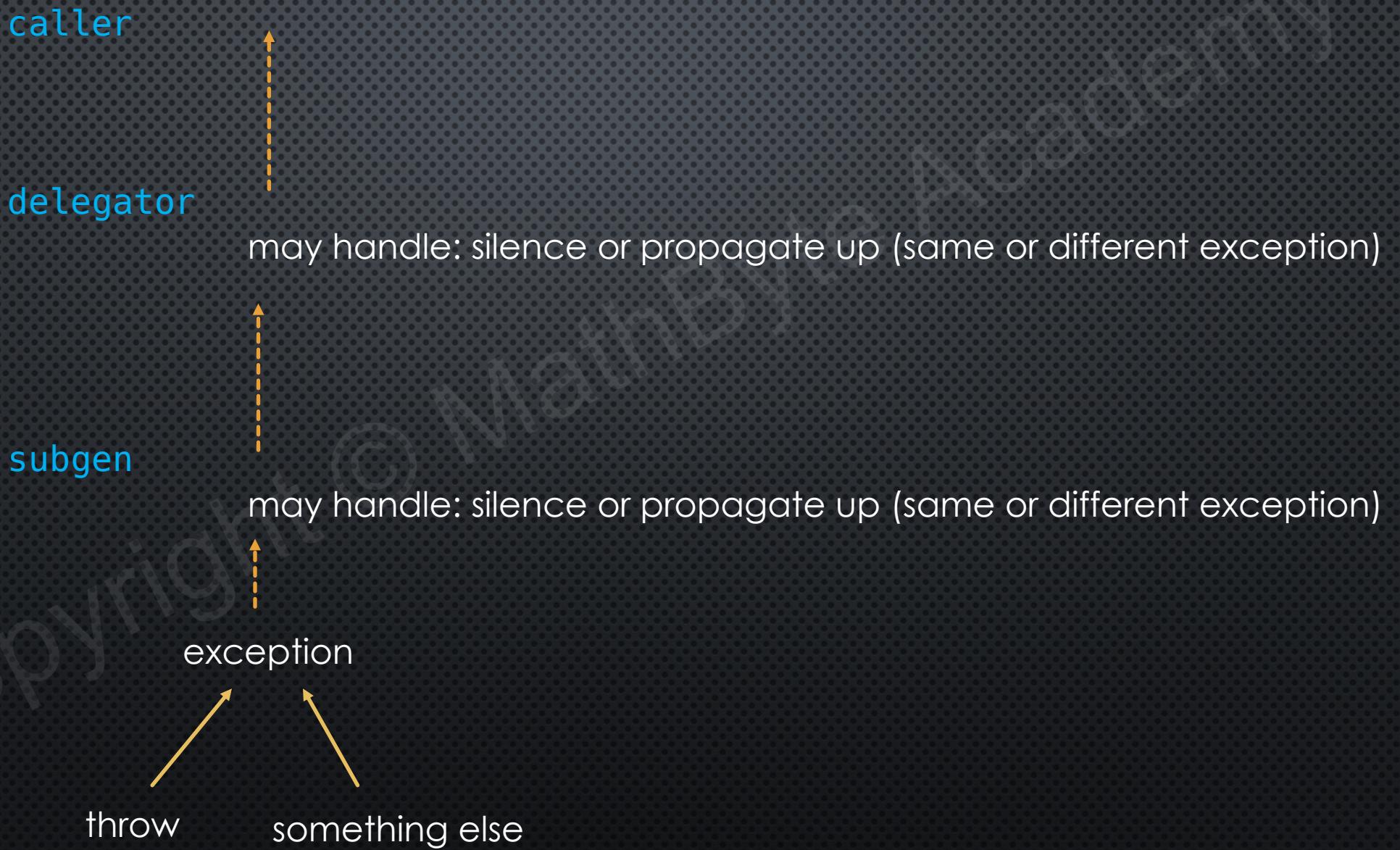


```
def subgen():  
    ...  
    yield  
    ...
```

Delegator **does not intercept** the exception → just forwards it to subgenerator

Subgenerator can then handle exception (or not)

Exception Propogation



PIPELINES

Copyright © 2014

Data pipelines (Pulling)



We've seen this before

→ use `yield` and iteration to pull data through the pipeline

consumer
iterate `filter_data()`
write data to file

`filter_data`
iterate `parse_data()`
`yield` select rows only

`parse_data`
iterate `read_data()`
transform data
`yield row`

`read_data`
`yield row`

pull

pull

pull

Data pipelines (Pushing)

With coroutines, we can also push (**send**) data through a pipeline



Example



Can get crazier...

broadcasting



pushes data through the pipeline