SEENG 265

Even more Python

Topics

- Python data model
- Decorators
- Generators
- `itertools`
- Coda: `__setitem__`

Python data model

- We have seen some of the special method names used by Python
    - `__repr__`
    - `__str__`
- There are many others, and these permit our code to implement, support, and interact with such things as:
    - iteration; collections; attribute access;
    - operator overloading; function and method invocation;
    - object creation and destruction;
    - string representation and formatting
    - managed contexts

- The data model, however, may seem strange if approached from knowledge of Java.
- When Java-ness is set aside, and Python-nese is embraced, this helps encourage a style of coding called **Pythonic**.
- To unpick this a little bit, let's look at an example involving two *special methods*
    - `__getitem__`
    - `__len__`
- Our example will be a **card deck**
- We'll also introduce the concept of a **named tuple** (a useful cross between a tuple and a dictionary)

In [ ]:
```
import collections    # Part of base Python

Card = collections.namedtuple('Card',
                              ['rank', 'suit'])

class FrenchDeck:
    # No constructor!

    def __len__(self):
        return 52

    def __getitem__(self, position):
        if position >= 52:
            raise IndexError

        suit_index = position // 13
        rank_index = (position % 13) + 2

        if (suit_index == 0):   suit = "spades"
        elif (suit_index == 1): suit = "hearts"
        elif (suit_index == 2): suit = "diamonds"
        else:                   suit = "clubs"

        if rank_index >= 2 and rank_index <= 10:
            rank = rank_index
        elif rank_index == 11: rank = 'J'
        elif rank_index == 12: rank = 'Q'
        elif rank_index == 13: rank = 'K'
        elif rank_index == 14: rank = 'A'

        return Card(str(rank), str(suit))
```

In [ ]:
```
devils_bedpost = Card('4', 'clubs')
devils_bedpost
```

In [ ]:

```
# Because of __getitem__ we can use [] to
# index an item

deck = FrenchDeck()
print(deck[0])
print(deck[-1])
```

In [ ]:

```
from random import choice

print("First pick:                        ", choice(deck))
print("Another pick from whole deck:", choice(deck))
```

In [ ]:

```
Card('Q', 'hearts') in deck
```

In [ ]:

```python
# "Uno" card deck

Card('green', '4') in deck
```

In [ ]:

```
# Because of __len__ and __getitem__, we can use a deck
# in a for-loop

for card in deck:
    print(card)
```

In [ ]:

```
for card in reversed(deck):
    print(card)
```

- With the use of `__len__` and `__getitem__`, we get lots of useful Python behavior for free!
    - `[]` based access of elements
    - `in` operator
    - iteration through a `FrenchDeck` object

- However, we quickly run into some limits
- For example:
    - Can we take the first three cards off the pile?

In [ ]:
```
deck[:3]
```

- No.
- We would need to implement slicing behavior in our class.

- Let us re-write the card deck
- We'll be a bit more clever now with our implementation, and exploit the power of lists.

In [ ]:

```
import collections   # Part of base Python

Card = collections.namedtuple('Card', ['rank', 'suit'])

class AnotherFrenchDeck:    # The card deck you know and love was invented in Fran
ce
    ranks = [str(n) for n in range(2, 11)] + list("JQKA")
    suits = "spades hearts diamonds clubs".split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

In [ ]:

```
deck = AnotherFrenchDeck()
print(deck[0])
print(deck[-1])

print("-" * 40)
from random import choice
print(choice(deck))
print(choice(deck))
```

In [ ]:

```
for card in deck:
    print(card)
```

In [ ]:

```
for card in reversed(deck):
    print(card)
```

In [ ]:

```
Card('Q', 'hearts') in deck
```

In [ ]:
```python
# "Uno" card deck

Card('green', '4') in deck
```

In [ ]:
```python
deck[:3]
```

- We could also sort the cards
    - Let's sort first by rank (aces high) and then by suit (spades high)
    - This is a bit different from how we often sort a playing deck
    - Our key function will convert a card (rank + suit) into a number
    - Therefore 2 of `Clubs` comes before a 2 of `Diamonds`, which all come before a 3 of `Clubs`, ..., with `A` of `hearts` and `A` of `Spades` at the end of sorted order.

In [ ]:
```
suit_values = { "spades":3, "hearts":2, "diamonds":1, "clubs":0 }

def card_key(card):
    rank_value = AnotherFrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

In [ ]:
```
for card in sorted(deck, key=card_key):
    print(card)
```

- Can we shuffle?

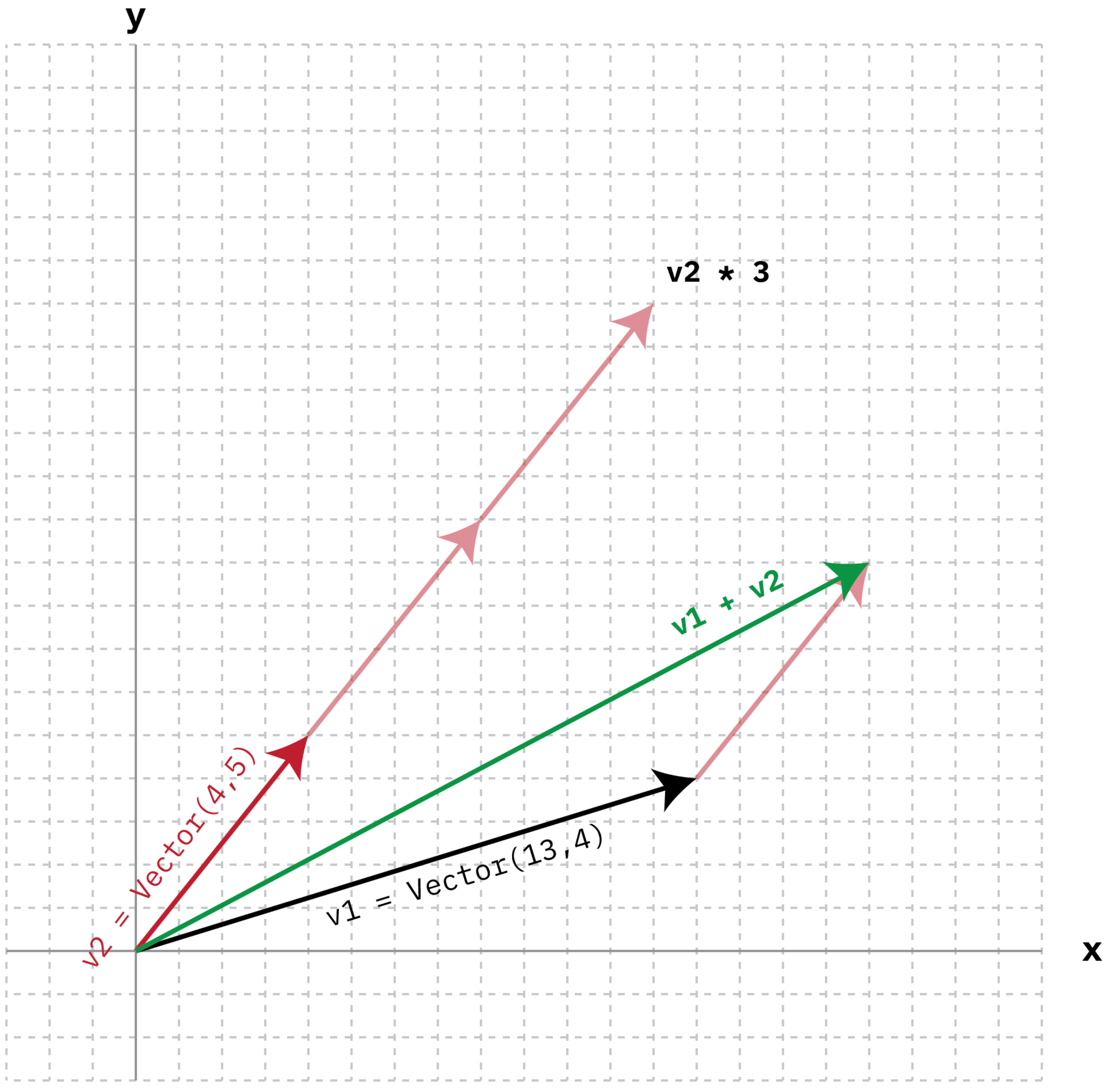
```
In [ ]: import random

        demo = list(range(10))
        random.shuffle(demo)
        demo
```

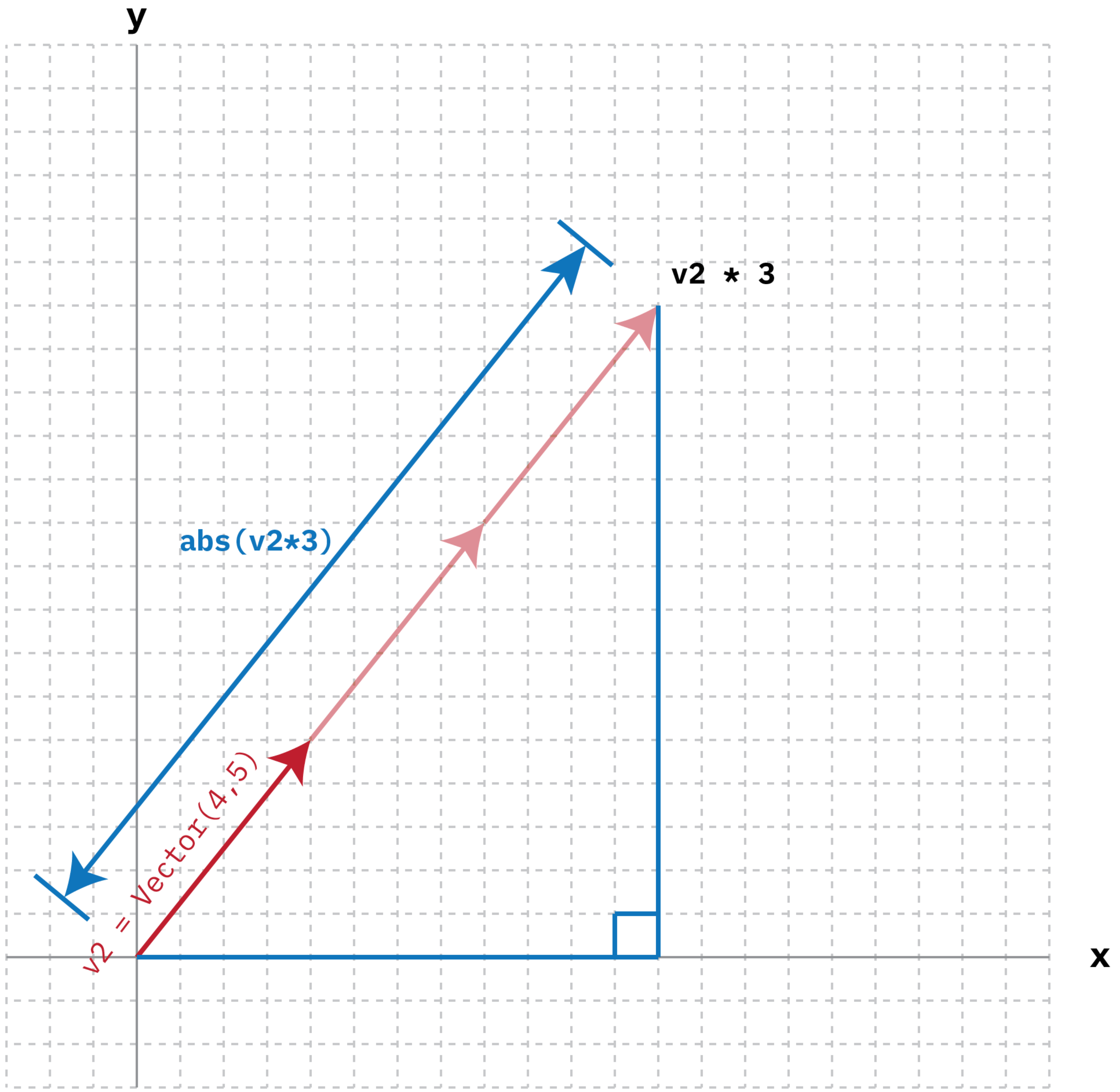
```
In [ ]: random.shuffle(deck)
        deck
```

- The problem is the `AlternateFrenchDeck` appears to be immutable
  - Later we'll see how to fix this via the `__setitem__` special method.

- The important points to understand here:
  - Python supports classes ...
  - ... but does not use them like Java does to indicate inherited functionality.
  - Rather: the Python data model permits objects to be used **as they are at runtime** ...
  - ... so if that object has `__len__` and `__getattr__` attributes, then that object can be used in ways available through those special methods

- Another example: a two-dimensional `Vector` class
- We'd like to have:
    - expressions such as `Vector(13, 4)` to create vectors
    - use the + operator to add vectors together (i.e., new vector with corresponding components added)
    - obtain the magnitude of a vector
    - use a scalar multiplier on a vector


y
x
v2 * 3
v1 + v2
v2 = Vector(4,5)
v1 = Vector(13,4)


y
x
v2 * 3
abs(v2*3)
v2 = Vector(4,5)

In [ ]:
```
import math

class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Vector(%r, %r)' % \
            (self.x, self.y)

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar,
                      self.y * scalar)
```

In [ ]:
```
v1 = Vector(13, 4)
v2 = Vector(4, 5)
v1 + v2
```

In [ ]:
```
abs(v1)
```

In [ ]:
```
v2 * 3
```

- Special method names that **do not involve** operator symbols (not an exhaustive list)

category	**method names**
string/bytes representation	`__repr__`, `__str__`, `__format__`, `__bytes__`
emulating collections	`__len__`, `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`
instance creation and destruction	`__new__`, `__init__`, `__del__`
conversion to number	`__abs__`, `__bool__`, `__complex__`, `__int__`, `__float__`, `__hash__`
etc.	etc.

- Special method names that **do** involve operator symbols (not an exhaustive list)

category	method names
unary numeric operators	`__neg__` **-**, `__pos__` **+**, `__abs__` **`abs()`**
rich comparison operators	`__lt__` **<**, `__le__` **<=**, `__eq__` **==**, `__ne__` **!=**, (etc.)
arithmetic operators	`__add__` **+**, `__sub__` **-**, `__mul__` *****, `__truediv__` **/**, `__floordiv__` **//**, (etc.)
augumentic assignment arithmetic operators	`__iadd__`, `__isub__`, `__imul__`, `__itruediv__`, `__ifloordiv__`, (etc.)
etc.	etc.

- See: `https://docs.python.org/3.7/reference/datamodel.html`

Decorators

- For many programmers -- even those with much experience using Python -- **decorators** can seem mysterious
- Part of this is because there are several dimension to decorators:
    1. The syntax necessary **to use** a decorator
    2. **Implementing a decorator from scratch** (including learning some new syntax)
    3. The typical use-cases involving **best practices when using the Python library**

- Unfortunately we cannot get very deep into decorators in these lectures ...
- ... but what follows here is meant to give you a stronger intuition of what is going on ...
- ... such that when you begin to use decorators in the Python library, the result will seem much less mysterious.

- Some review/context: **What do we achieve by writing and using functions?**

```
...
operation W
operation X
...

...
operation Y
operation Z
...

...
operation Y
operation Z
...

...
operation W
operation X
...

...
operation Y
operation Z
...
```

```
...
function_WX()
...

...
function_YZ()
...

...
function_YZ()
...

...
function_WX()
...

...
function_YZ()
...
```

```
def function_WX():

    operation W
    operation X
```

```
def function_YZ():

    operation Y
    operation Z
```

operation A
function_WX()
operation B

operation A
function_YZ()
operation B

operation A
function_YZ()
operation B

operation A
function_WX()
operation B

operation A
function_YZ()
operation B

```
def function_WX():

    operation W
    operation X
```

```
def function_YZ():

    operation Y
    operation Z
```

```
function_WX()

function_YZ()

function_YZ()

function_WX()

function_YZ()
```

```
@decorator_AB
def function_WX():
    operation W
    operation X
```

```
@decorator_AB
def function_YZ():
    operation Y
    operation Z
```

```
def decorator_AB(f):
    operation A
    <use f somehow>
    operation B
```

- The intuition behind decorators is therefore:
    - A **decorator** is meant to perform some actions **around** some specified function (which becomes the **decorated function**).
    - **Tricky bit is recognizing how that given function is called within the decorator...**
    - ... plus how the decorator ensures the function receives its needed arguments.

- Example (without decorators): Adding some timing code around existing queries for some user information

In [ ]:

```python
def obtain_name(greeting):
    n = input(greeting + " What is your name? ")
    return n

def obtain_age():
    while (True):
        try:
            age = input("What is your age (in years)? ")
            age = int(age)
            return age
        except ValueError:
            print("Sorry, we need a whole number. Try again.")

def main():
    n = obtain_name("Welcome to Bluefish Insurance Brokers.")
    a = obtain_age()
    print(n, "is", a, "years of age. Woot, I say.")

main()
```

- Suppose now we want to somehow capture the time taken to provide input?
- That is:
    - Need to determine the current time before `obtain_name` or `obtain_age` are called...
    - ... then perform the function itself (i.e. either `obtain_name` or `obtain_age`) ...
    - ... then determine the current time when the function is finished ...
    - ... and finally report the difference (in a human-readable form).

In [ ]:

```
import time

def duration(func):
    def stopwatch(*args):
        t0 = time.perf_counter()
        result = func(*args)
        elapsed = time.perf_counter() - t0
        print('[%0.8fs]' % (elapsed))
        return result
    return stopwatch
```

- Line 3: Name of decorator, with function to be decorated to be acccessed using `func` parameter.
- Line 4: `stopwatch` is a **nested function** -- but there is no intention for any other code in the program to *directly* called `stopwatch` by name.
- Also line 4: `*args` gets access to all positional arguments passed to `stopwatch` (explained in a moment!)

- Line 5: Current time
- Line 6: Call the function that has been generated (`func`), passing all arguments given to `stopwatch` to `func`
- Lines 7 & 8: Current time, compute duration, print result

In [ ]:

```
import time

def duration(func):
    def stopwatch(*args):
        t0 = time.perf_counter()
        result = func(*args)
        elapsed = time.perf_counter() - t0
        print('[%0.8fs]' % (elapsed))
        return result
    return stopwatch
```

- Line 9 and 10: The tricky parts!
    - But let's explain these after we see the resulting code.

In [ ]:

```
@duration
def obtain_name(greeting):
    n = input(greeting + " What is your name? ")
    return n

@duration
def obtain_age():
    while (True):
        try:
            age = input("What is your age (in years)? ")
            age = int(age)
            return age
        except ValueError:
            print("Sorry, we need a whole number. Try again.")

def main():
    n = obtain_name("Welcome to Bluefish Insurance Brokers.")
    a = obtain_age()
    print(n, "is", a, "years of age. Woot, I say.")

main()
```

In [ ]:

```
# ....

def duration(func):
    def stopwatch(*args):
        t0 = time.perf_counter()
        result = func(*args)
        elapsed = time.perf_counter() - t0
        print('[%0.8fs]' % (elapsed))
        return result
    return stopwatch

# ....
```

- Line 10: This line returns a function that is also known as a **closure**
    - All uses of this return value have the meaning of `func` in `stopwatch` already embedded into a bundle of heap memory with an instance of `stopwatch`.
    - Now even though we think we're calling `obtain_name` or `obtain_age` or `main`, we're *really* calling `stopwatch`, but where `stopwatch` has additional state to know which of the three functions to call.

In [ ]:

```python
# @duration
def obtain_name(greeting):
    n = input(greeting + " What is your name? ")
    return n

@duration
def obtain_age():
    while (True):
        try:
            age = input("What is your age (in years)? ")
            age = int(age)
            return age
        except ValueError:
            print("Sorry, we need a whole number. Try again.")

def main():
    n = obtain_name("Welcome to Bluefish Insurance Brokers.")
    a = obtain_age()
    print(n, "is", a, "years of age. Woot, I say.")

main()
```

In [ ]:

```python
print(obtain_name)
print(obtain_age)
print(main)
```

In [ ]:

```
b = lambda x : x * 2
print( b("aardvark") )
print( duration(b)("timmy") )
```

- Decorators are usually brought out as a solution to some standard problems
    - That is, some already-existing functions in user code needs some additional properties
    - Rather than re-write the function, we add the properties via a decorator
- Library module `functools`
    - See `https://docs.python.org/3.6/library/functools.html`

- Your first use of decorators with regular code will probably end up as a form of "Harry Potter" programming:
    - Library author informs you how to call functions in the library.
    - Author may also suggest how to use decorators to add what you might need
- Decorators can also be **stacked**:
    - Decorating a decorated function

- So now you know what decorators look like...

Generators

- Before we look at generators, let's look at something much simpler: a `Sentence` class

In [ ]:
```
import reprlib

class Sentence:

        def __init__(self, text):
            self.text = text
            self.words = text.split()

        def __getitem__(self, index):
            return self.words[index]

        def __len__(self):
            return len(self.words)

        def __repr__(self):
            return 'Sentence(%s)' % reprlib.repr(self.text)
```

In [ ]:
```
s = Sentence("Shall I compare thee to a summer's day?")
s
```

In [ ]:
```
for word in s:
    print(word)
```

In [ ]:
```
list(s)
```

- All sequences in Python are ***iterable***
    - And they are iterable because all sequences implement the `__iter__` function...
    - ... either as a built-in (i.e., part of the Python interpreter) ...
    - ... or explicitly written (i.e., in the class)

- That is, when Python needs to iterate of an object `x`, it calls `iter(x)` on the object:
    - And this returns the result of the `__iter__` method if it has been implemented in the object's class...
    - ... and if this isn't the case, but `__getitem__` is implemented, then Python ***creates*** an iterator using `__getitem__` (i.e., to proceed from item 0 up to the number of items) ...
    - ... and if ***this*** isn't the case, then Python raises a `TypeError`.

Now this begins to get subtle...

- There is a different between **iterable** and **iterators**
    - Is it a little bit like the difference between an **interface** and an **implementation**.

- **iterable** is a property/characteristic possessed by some object.

- an **iterator** is method/code that implements two methods:
    - `__next__`: returns next available item, raising `StopIteration` when there are no more items
    - `__iter__`: returns `self` (i.e., for cases when object is used by a `for` loop and an iterable is expected.

In [ ]:

```python
s = 'ABC'
for char in s:
    print(char)
```

In [ ]:

```python
s = 'ABC'
it = iter(s)
while True:
    try:
        print(next(it))
    except StopIteration:
        del it
        break
```

- On right:
    - line 2 creates an iterable for the sequence `ABC`
    - line 5 fetches the `next` item in the sequence
    - lines 6, 7, and 8 are needed to stop the loop (and line 7 discards the iterator object)

- Let us re-write `Sentence` to use a custom iterator.
- Note: This is not necessarily the best practice (i.e., we should use what is built-in to Python as much as possible)...
- ... but we want to expose some of the machinery of an iterator as this will help make sense of generators.

In [ ]:

```python
import reprlib

class Sentence:

        def __init__(self, text):
            self.text = text
            self.words = text.split()
        """
        def __getitem__(self, index):
            return self.words[index]

        def __len__(self):
            return len(self.words)
        """

        def __repr__(self):
            return 'Sentence(%s)' % reprlib.repr(self.text)

        # New code
        def __iter__(self):
            return SentenceIterator(self.words)
```

In [ ]:

```
class SentenceIterator:

    def __init__(self, words):
        self.words = words
        self.index = 0

    def __next__(self):
        try:
            word = self.words[self.index]
        except IndexError:
            raise StopIteration()          # needed by Python to terminate a `for`
loop
        self.index += 1                    # update our current "position" in the
 sequence
        return word

    def __iter__(self):
        return self
```

In [ ]:

```
s = Sentence("Shall I compare thee to a summer's day?")
for word in s:
    print(word)
```

- Our `SentenceIterator` is used by the Python-interpreter's machine for a `for`
    - That is, everything the `for` machinery requires is provided by the iterator
- However, recall what that machinery needs:
    - An object with a `__next__` method that returns the next word

- Let us rewrite `Sentence` in a way that does **not** use a separate iterator class...
- ... but rather **generates an object that itself has a `next` operation**.

- Before we do that, however, let's look at the Python feature that will make this happen: the `yield` statement.

In [ ]:
```
def gen_ABC():
    yield 'A'
    yield 'B'
    yield 'C'
```

In [ ]:
```
gen_ABC
```

In [ ]:
```
g = gen_ABC()
next(g)
```

In [ ]:
```
next(g)
```

In [ ]:
```
next(g)
```

In [ ]:
```
next(g)
```

- The **big idea** here
  - `yield` produces a **generator** object
  - When code calls `next()` on such an object, Python resumes the generator from the statement after the previous `yield`, and returns the value with the upcoming `yield` as the result of `next()`.
  - (That is, the `yield` statement **yields a value** or **produces a value**)
  - Special cases are the first call to `next()`...
  - ... and when `next()` resumes at a point in the code where there **is nothing** following the previous yield (i.e., the function that created the generator is complete).

- That is:
  - **generators** are also **iterators**

In [ ]:

```python
import reprlib

class Sentence:

        def __init__(self, text):
            self.text = text
            self.words = text.split()

        def __repr__(self):
            return 'Sentence(%s)' % reprlib.repr(self.text)

        # New code
        def __iter__(self):
            for word in self.words:
                yield word
            return

        # That's all!!
```

In [ ]:

```python
s = Sentence("Shall I compare thee to a summer's day?")
it = s.__iter__()
next(it)
```

In [ ]:

```python
s = Sentence("You are my sunshine, my darling sunshine.")
for w in s:
    print(w)
```

In [ ]:

```python
next(it)
```

- And:
    - We can use a generator to implement a sequence that is infinitely long!
    - In practice, we evaluate that sequence **lazily** (i.e., only calculate as much lf the sequence as is needed in some step)

- Syntactic point for code that follows:
    - We can determine the type of a value at runtime using the `type` built-in
    - This built-in returns an `class` object.

In [ ]:

```
m = 10
f = 10.0
print(type(m))    # print the type-class returned for value of m
print(type(f))    # print the type-class returned for value of f
```

- With the class object for some numeric value A, we can then convert some other number B that into the numeric type of A.

In [ ]:

```
f_converted = type(m)(f)  # type(m) is an object; type(m)(f) calls a type-construc
tor with value of f
m_converted = type(f)(m)  # type(f) is an object; type(f)(m) calls a type-construc
tor with value of m
print(type(f_converted))
print(type(m_converted))
```

- Notice the loop below from lines 12 to 15:

In [ ]:
```
class ArithmeticProgression:

    def __init__(self, begin, step, end=None):
        self.begin = begin
        self.step  = step
        self.end = end # None implies an
                       # "infinite sequence/series"

    def __iter__(self):
        result = type(self.begin + self.step)(self.begin)
        forever = self.end is None
        index = 0
        while forever or result < self.end:
            yield result
            index += 1
            result = self.begin + self.step * index
```

In [ ]:
```
cardinal = ArithmeticProgression(0, 1.1)
# list(cardinal)
cc = iter(cardinal)
for _ in range(10000000):
    p = next(cc)
print(p)
```

In [ ]:
```
vals = cc[:10]
```

- The `yield` statement is short yet surprisingly powerful.
- The hard part is to accept that the use of the `next()` built-in function will cause the generator to resume from the statement following the `yield`

- The ability to create an infinite series is delightful...
- ... but is not the main purpose of a generator ...
- ... although when you see the use of an infinite sequence (lazily evaluated) then having this mechanism results in a robust and extensible solutions.

`itertools`

- Sometimes we may have a sequence or collection over which we want to iterate
- We may also want **select**, **compute**, or **rearrange** as we iterate.

- Python provides a rich library of **generator** functions that help with this
  - Idea: Given a sequence `seq` plus some function `f` that works on a sequence item ...
  - ... the generator function applies `f` in a particular way on items in `seq`.

- The easiest way to explain this is to show with some examples

`itertools`: filtering

- Assume the following function that we define:

In [ ]:
```python
# returns True or False

def vowel(c):
    return c.upper() in 'AEIOU'
```

- There is already a built-in function named `filter` which can use a function such a `vowel`:

In [ ]:
```python
list(filter(vowel, "eelgrass"))
```

- Here are some generator functions in `itertools` that can be used for **filtering**:

In [ ]:
```
import itertools

itertools.filterfalse(vowel, "eelgrass")
```

In [ ]:
```
list(itertools.filterfalse(vowel, "eelgrass"))
```

In [ ]:
```
list(itertools.takewhile(vowel, "eelgrass"))
```

In [ ]:
```
list(itertools.dropwhile(vowel, "eelgrass"))
```

In [ ]:
```
list(itertools.compress("eelgrass", [1, 0, 1, 1, 0, 1, 1]))
```

`itertools`: computation

- Here are some generator functions that perform computation along with **mapping**:

In [ ]:
```
data = [31, 41, 59, 26, 53, 58, 97, 93, 23, 84, 62, 6, 43]

import itertools

list(itertools.accumulate(data))   # default operation is `+`: running sum
```

In [ ]:
```
list(itertools.accumulate(data, min))   # running minimum value
```

In [ ]:
```
list(itertools.accumulate(data, max))   # running maximum value
```

In [ ]:
```
import operator

list(itertools.accumulate(data, operator.mul))  # running product
```

In [ ]:
```
list(itertools.accumulate(range(1,11), operator.mul))   # factorials from 1! to 1
0!
```

In [ ]:
```
# map is a powerful general-purpose generator
# Notice two lists (parameters 2 and 3)

list(map(operator.mul, range(12), range(12)))   # squares from 0 to 11
```

In [ ]:
```
# starmap does some similar
# Notice the list as the second parameter (a list of duples)

list(itertools.starmap(operator.pow, [(1, 3), (4, 9), (12, 0.5), (5, -2)]))
```

`itertools`: merging

- You may have seen these described in Python code-examples that you have seen from web searches:

In [ ]:
```
# Given two sequences, create duples
# (limited by shortest sequence)

list(zip('ABC', range(5)))     # Given two sequences, create duples (lim)
```

In [ ]:
```
# Given three sequences, create triples
# (limited by shortest sequence)

list(zip('ABC', range(5), ["u", "v", "w", "x", "y", "z"]))
```

In [ ]:
```
# Given three sequences, create triples
# (if some sequences are too short, substitute "???" where necessary)

list(itertools.zip_longest('ABC', range(5), ["u", "v", "w", "x", "y", "z"], fillva
lue="???"))
```

`itertools`: product

- Some of these can be already expresses as list comprehensions...
- ... but the generator-nature of `itertools` means the result sequences are built lazily (i.e., only when needed)

In [ ]:
```
list(itertools.product('PQR', range(2)))     # Cartesian product
```

In [ ]:
```
suits = "spades hearts diamonds clubs".split()

list(itertools.product('AK', suits))     # Another Cartesian product
```

In [ ]:
```
list(itertools.product('XYZ'))    # Looks a little odd, but there is some logic her
e.
```

In [ ]:

```
# `repeat=<N>` means "consume each iterable <N> times"

list(itertools.product('XYZ', repeat=2))
```

In [ ]:

```
# `repeat=<N>` means "consume each iterable <N> times"
list(itertools.product(range(2), repeat=3))
```

`itertools`: much more

- There is a lot of functionality in this module.
- The point of these slides is not to simply present one d*mn thing after another!

- Rather:
    - Sometimes we can obtain in one line what would normally require many other lines of Python.
    - Perhaps will spend as much time thinking about the immediate problem we're trying to solve...
    - ... but by writing less code, we increase the program's accuracy and extensibility.

Last two slides

- Remember how `AnotherFrenchDeck` could not be shuffled?

In [ ]:

```
import collections   # Part of base Python

Card = collections.namedtuple('Card', ['rank', 'suit'])

class AnotherFrenchDeck:    # The card deck you know and love was invented in Fran
ce
    ranks = [str(n) for n in range(2, 11)] + list("JQKA")
    suits = "spades hearts diamonds clubs".split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

In [ ]:

```
import random

deck = AnotherFrenchDeck()
print(deck[:3])
random.shuffle(deck)
```

- We could modify the class definition to add `__setitem__`...
- .. or we could perform a **monkey patch** where we simply add the method directly to the class outside the definition!

In [ ]:
```
def set_card(deck, position, card):
    deck._cards[position] = card

AnotherFrenchDeck.__setitem__ = set_card
```

In [ ]:
```
random.shuffle(deck)
print(deck[:5])
```

In [ ]:
```

```

- Cool, *n'est pas?*
- (*oder, "Kul, nicht wahr?"*)

Summary

- Python's power and flexibility is a combination of:
    - A carefully thought-out data model
    - A rich library of operators and functions that use this data model
    - A class mechanism that is not over encumbered by interface features (as is the case with Java)
- Getting use to this combination takes a bit of experience and effort
    - These lectures are really meant to open a window a bit wider for you...