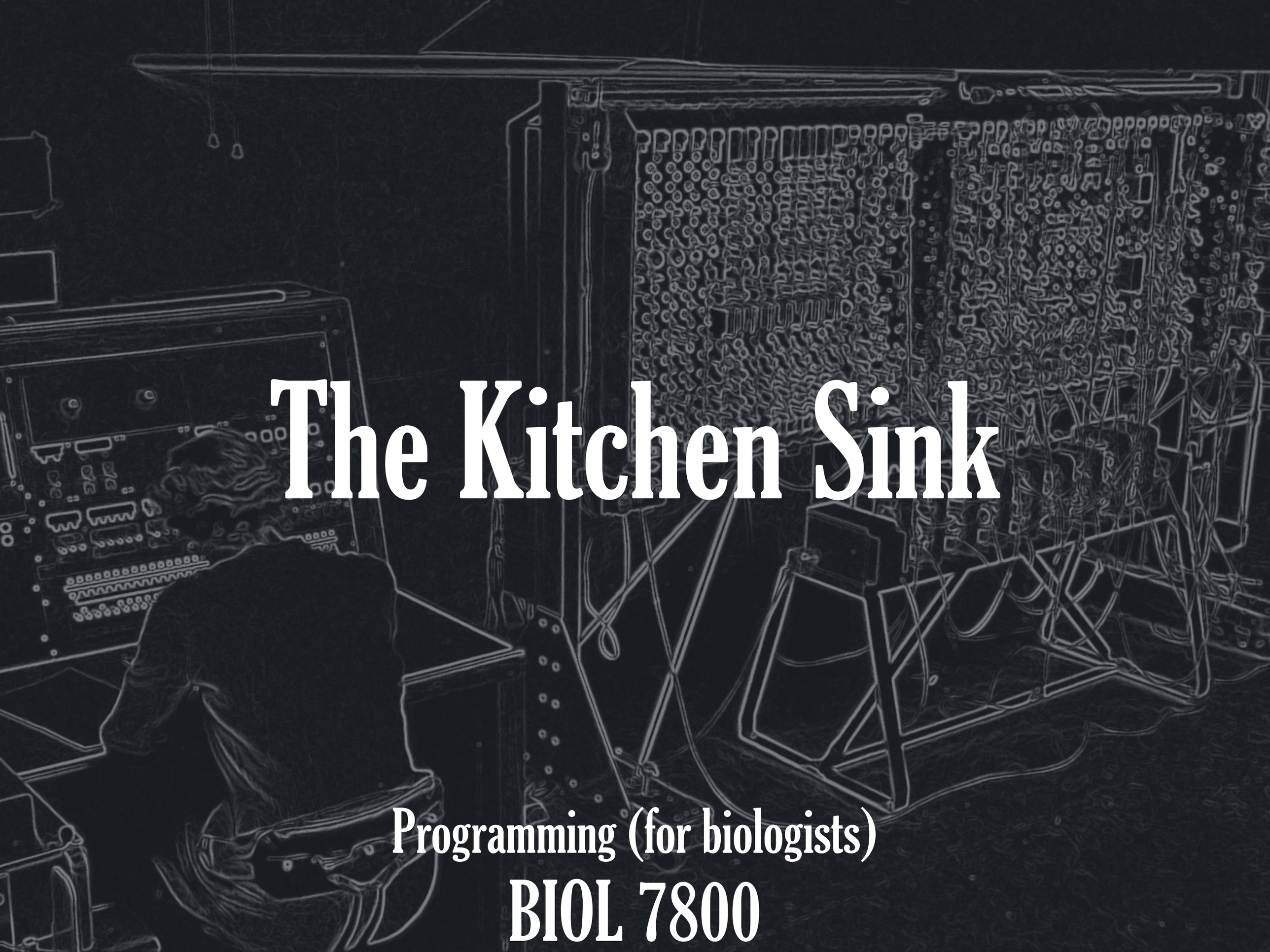


The Kitchen Sink

Programming (for biologists)

BIOL 7800

map()

example14.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example14.py

```
1   #!/usr/bin/env python
2
3
4   def power(data):
5       result = []
6       for num in data:
7           result.append(num**2)
8       return result
9
10
11  def main():
12      nums = range(0, 10)
13      result = power(nums)
14      print(result)
15
16  if __name__ == '__main__':
17      main()
18
```

File 0 | Project 0 | ✔ No Issues | example14.py | 14:17 | LF | UTF-8 | Python

"Standard" way to raise all items of a list to a certain power

```
% python example13.py
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

map()

example14.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example14.py

```
1   #!/usr/bin/env python
2
3
4   def power(data):
5       return data**2
6
7
8   def main():
9       nums = range(0, 10)
10      result = map(power, nums)
11      print(list(result))
12
13  if __name__ == '__main__':
14      main()
15
```

map() applies a function to every item of an iterable

function

map() returns an iterable where function has been run on every item

iterable

```
% python example13.py
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

File 0 | Project 0 | ✔ No Issues | example14.py | 15:1 | LF | UTF-8 | Python

iterators

Python 2.7

```
In:  type(range(0, 10))
Out: list
In:  print(range(0, 10))
Out: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

list

Python 3.5

```
In:  type(range(0, 10))
Out: range
In:  print(range(0, 10))
Out: range(0, 10)
```

iterable

An object that has an `__iter__` method

Returns an `iterator`

iterators

What is an iterator?

It's an object with a `__next__` method
that returns a single item of the object

Now, there's a couple of ways you can do this....

example103.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example103.py

```
1  class Reverse:
2      """Iterator for looping over a sequence backwards."""
3      def __init__(self, data):
4          self.data = data
5          self.index = len(data)
6      def __iter__(self):
7          return self
8      def __next__(self):
9          if self.index == 0:
10             raise StopIteration
11         self.index = self.index - 1
12         return self.data[self.index]
13
```

Define an `__iter__` method

Define a `__next__` method

iterators

my_string = 'abcdefg'

my_list = [1, 2, 3, 4]

my_tuple = (1, 2, 3, 4)

Many types in Python are `iterable`

itertools module

Many types in Python are `iterable`

and `itertools` gives us ways of working with those iterators
these are fast and memory efficient

Infinite iterators	Terminate on shortest input	Combinatoric generators
count()	compress()	product()
cycle()	imap()	permutations()
repeat()	tee()	combinations()

itertools module

Infinite iterators

count()

provides an infinite count, starting at x

```
In:  for elem in count(10)
         print(elem)
Out: 10
Out: 11
Out: 11
Out: 13
...
```

cycle()

provides an infinitely repeating cycle

```
In:  for elem in cycle([1,2])
         print(elem)
Out: 1
Out: 2
Out: 1
Out: 2
...
```

repeat()

repeats an element endlessly (or up to x times)

```
In:  for elem in repeat('A', 100)
         print(elem)
Out: 'A'
Out: 'A'
Out: 'A'
Out: 'A'
...
```

itertools module

Iterators that terminate on the shortest input

compress()

selects data (d[0] if s[0]), (d[1] if s[1])

```
In:  list(compress([1,2,3,4,5], [0,1,0,1,0]))
Out: [2,4]
```

...

starmap()

applies a function to iterators
func(*args), func(*args)

```
In:  list(starmap(pow, [(2,3), (3,3)]))
Out: [8, 27]
```

tee()

makes n iterators

```
In:  for i in itertools.tee([1,2,3], 3):
         print(list(i))
Out: [1, 2, 3]
Out: [1, 2, 3]
Out: [1, 2, 3]
```

itertools module

Iterators that generate combinatorial output

product()

Cartesian product

```
In:  list(product('ABC', repeat=2))
Out:
[('A', 'A'),
 ('A', 'B'),
 ('A', 'C'),
 ('B', 'A'),
 ('B', 'B'),
 ('B', 'C'),
 ('C', 'A'),
 ('C', 'B'),
 ('C', 'C')]
```

permutations()

r-length tuples, all possible orderings,
no repeated elements

```
In:  list(permutations('ABC'))
Out:
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'),
 ('B', 'C', 'A'), ('C', 'A', 'B'), ('C', 'B', 'A')]
```

combinations()

r-length tuples, sorted order, no repeated elements

```
In:  list(combinations('ABC', 2))
Out: [('A', 'B'), ('A', 'C'), ('B', 'C')]
```

itertools module

https://docs.python.org/3.5/library/itertools.html

`itertools.`**`combinations`**(*iterable*, *r*)

Return *r* length subsequences of elements from the input *iterable*.

Combinations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

Equivalent to:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

The code for `combinations()` can be also expressed as a subsequence of `permutations()` after filtering entries where the elements are not in sorted order (according to their position in the input pool):

time and datetime modules

These are modules for operating with times and dates

`time` module

principally for working with unix time values

`datetime` module

for more "standard" date+time manpulations

time and datetime modules

These are modules for operating with times and dates

`time` module

principally for working with
unix time values

```
In:  import time
In:  t = time.time()
In:  print(t)
Out:1457978948.740969
In:  time.localtime(time.time())
Out:time.struct_time(
tm_year=2016, tm_mon=3,
tm_mday=14, tm_hour=13,
tm_min=11, tm_sec=18,
tm_wday=0, tm_yday=74,
tm_isdst=1)
In:  time.strptime("30 Nov 00", "%d %b %y")
Out:time.struct_time(tm_year=2000, tm_mon=11,
tm_mday=30, tm_hour=0,
tm_min=0, tm_sec=0,
```

← Represents time as seconds

← `localtime()` can convert that

← `strptime()` allows us to input arbitrar
dates+times

`strftime()` is it's opposite

time and datetime modules

These are modules for operating with times and dates

datetime module

for more "standard" date+time manipulations

Has **3** submodules

datetime.date()

```
In:  import datetime
In:  today = datetime.date.today()
In:  tomorrow = datetime.date(2016,3,15)
In:  tomorrow - today
Out:datetime.timedelta(1)
In:  next_year = datetime.date(2017,3,14)
Out:datetime.timedelta(365)
```

Reasonably useful

datetime.time()

```
In:  import datetime
In:  lunch = datetime.time(13,0,0)
In:  lunch.isoformat()
Out:'13:00:00'
```

Less useful

Hard to work with

time and datetime modules

These are modules for operating with times and dates

datetime module

for more "standard" date+time manipulations

datetime.datetime()

```
In:  import datetime
In:  today = datetime.datetime.today()
In:  today.isoformat()
Out:'2016-03-14T14:00:32.403901'

In:  lunch = datetime.datetime(2016, 3, 15, 12, 00)
Out:lunch.isoformat()

In:  new_lunch = lunch + datetime.timedelta(minutes=75)
Out: new_lunch.isoformat()

In:  new_lunch = lunch + datetime.timedelta(days=119)
In:  new_lunch.isoformat()
Out: '2016-07-12T12:00:00'
In:  new_lunch.weekday()
Out: 2
```

Change our lunch date by 75 minutes

Change our lunch date by 119 days

Monday = 0; Sunday = 6

time and datetime modules

These are modules for operating with times and dates

datetime module

for more "standard" date+time manipulations

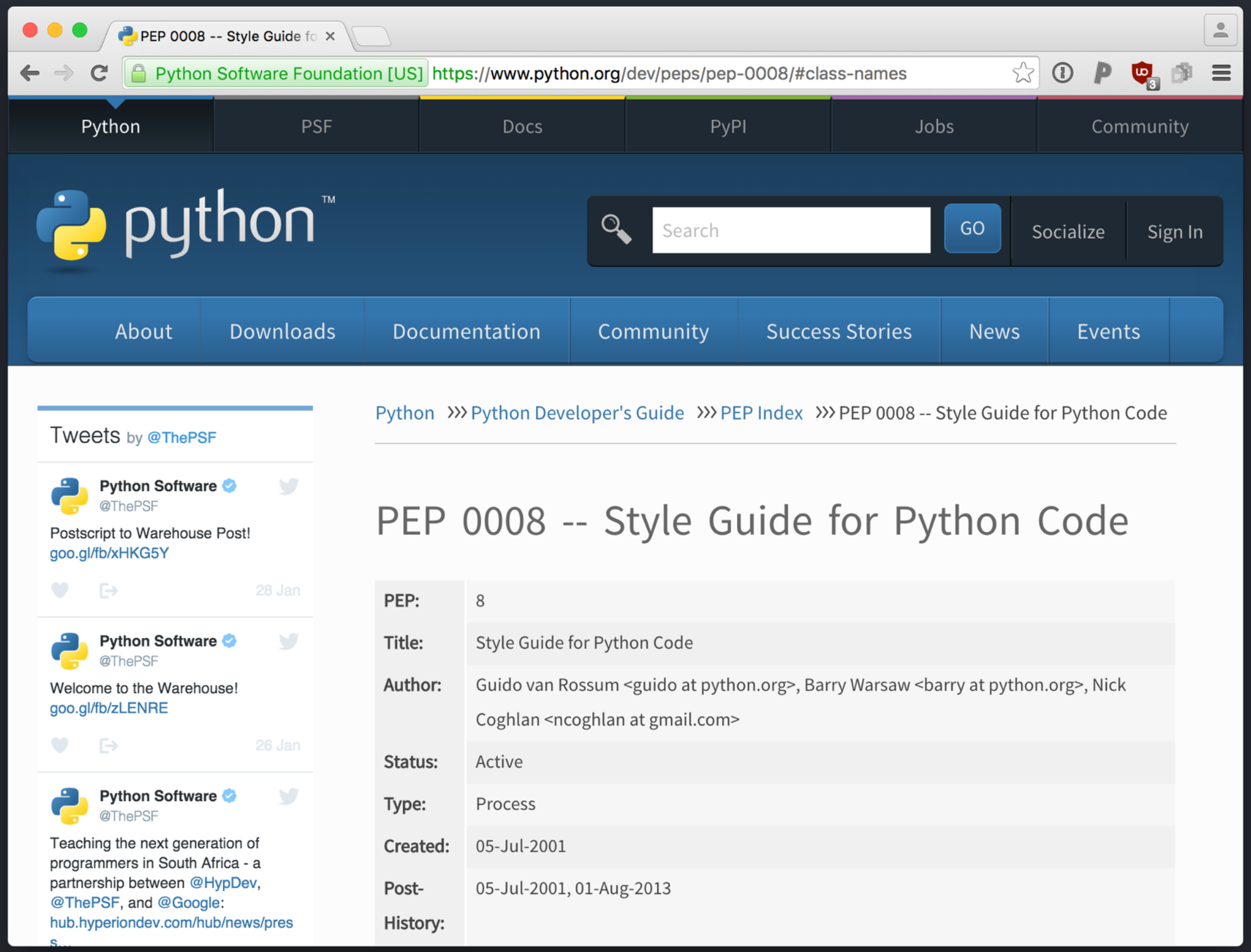
datetime.datetime()

```
In:  import datetime
In:  d1 = datetime.datetime(2016,1,1)
In:  d2 = datetime.datetime(2016,3,15)
In:  d1 - d2
Out:datetime.timedelta(-74)

In:  import datetime
In:  d1 = datetime.datetime(2016,1,1,13,5)
In:  d2 = datetime.datetime(2016,3,15,17,25)
In:  d1 - d2
Out:datetime.timedelta(-75, 70800)
```

number of seconds

PEP8 - Python Style Guide



Assignment 15.