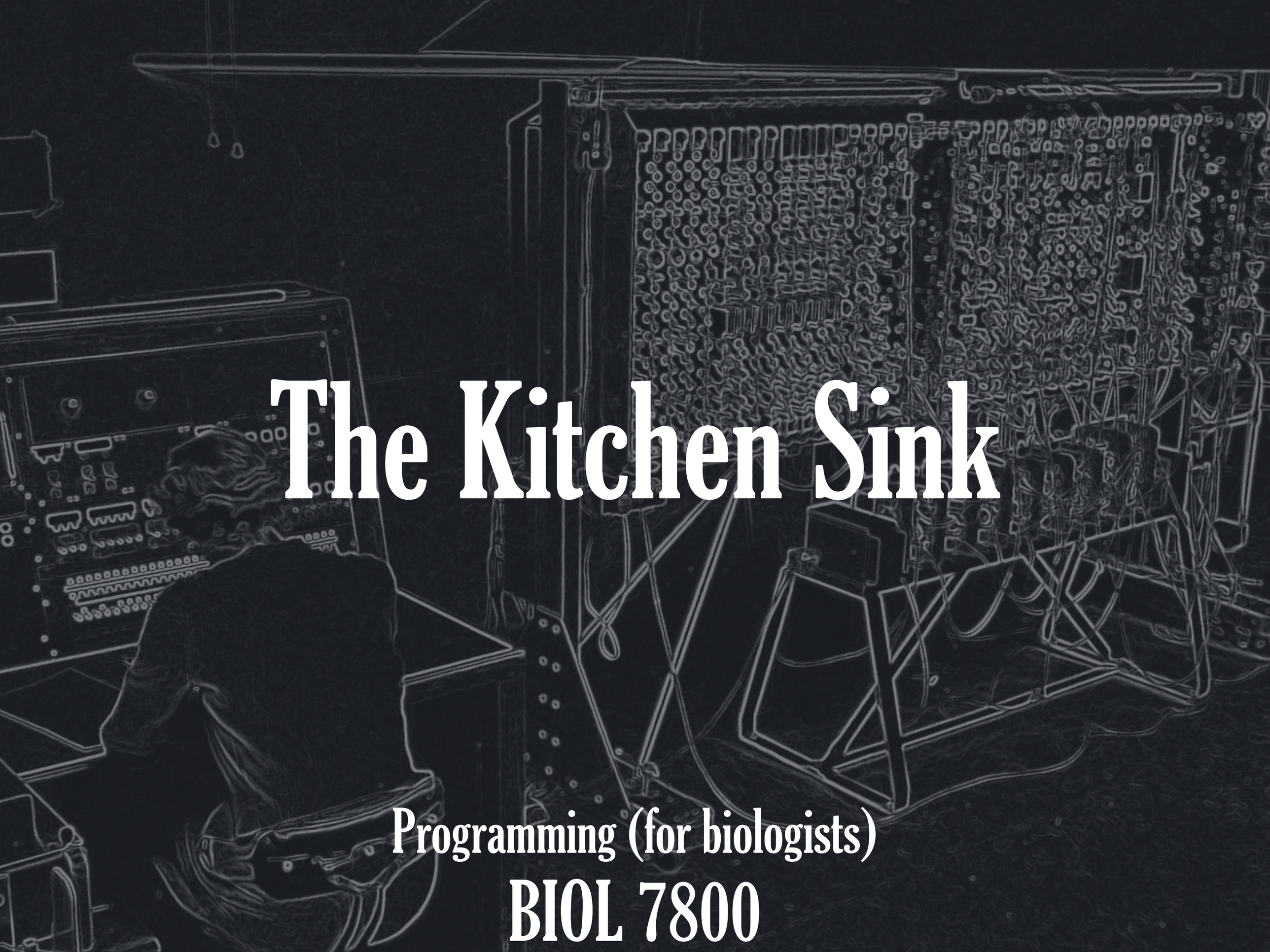


The Kitchen Sink

Programming (for biologists)

BIOL 7800

Sets

The last of the *major* Python `types`

A **set** is an unordered collection of **`unique`** elements

So, what has happened below?

set('able was I ere I saw elba')

{' ', 'a', 'b', 'e', 'i', 'l', 'r', 's', 'w'}

Sets

The last of the *major* Python types

A **set** is an unordered collection of **unique** elements

So, what has happened below?

```
set('able was I ere I saw elba')
```

```
{' ', 'a', 'b', 'e', 'i', 'l', 'r', 's', 'w'}
```

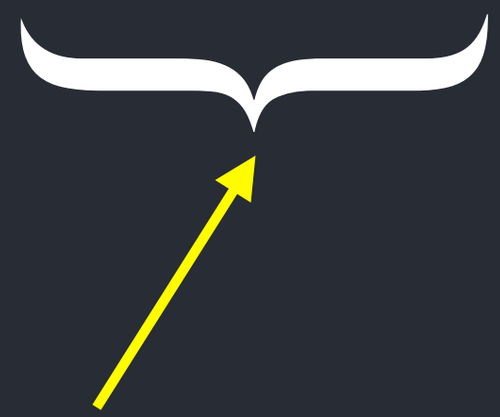
set() has unique-ified our string

Sets

Of course, we can also simply create sets

Basically like wrapping a `list` with `set()`

```
my_set = set([1, 2, 3, 4, 'dog'])
```



Just like a `list`, `sets()` can have *heterogenous content*

Sets

Of course, we can also simply create sets

But what about this?

integers string dict

```
my_set = set([1, 2, 3, 4, 'dog', {'thebest':'hotdogs'}])
```

Sets

Of course, we can also simply create sets

But what about this?

integers string dict

```
my_set = set([1, 2, 3, 4, 'dog', {'thebest':'hotdogs'}])
```

```
TypeError: unhashable type: 'dict'
```

Sets

Of course, we can also simply create sets

But what about this?

integers string list

```
my_set = set([1, 2, 3, 4, 'dog', [7, 8, 9]])
```

```
TypeError: unhashable type: 'list'
```

Sets

Of course, we can also simply create sets

But what about this?

integers string tuple

```
my_set = set([1, 2, 3, 4, 'dog', (7, 8, 9))
```

✓ Works

Sets

Of course, we can also simply create sets

But what about this?

integers string tuple

```
my_set = set([1, 2, 3, 4, 'dog', (7, 8, 9))
```

set() must contain what is known as a "hashable" object in Python and because objects like dicts and lists are mutable, they cannot be "hashed".

Sets

set() is very, very useful for unique-ifying lists

and performing set operations on the resulting unique sets

10 members

```
list1 = [
'locus-1',
'locus-2',
'locus-3',
'locus-3',
'locus-4',
'locus-5',
'locus-5',
'locus-6',
'locus-6',
'locus-6'
]
```

many items; non-unique

6 members

```
set1 = set([
'locus-1',
'locus-2',
'locus-3',
'locus-4',
'locus-5',
'locus-6',
])
```

unique items

Sets

set() operations

```
set1 = set([
  'locus-1',
  'locus-2',
  'locus-3',
  'locus-4',
  'locus-5',
  'locus-6'
  ])
```

```
set2 = set([
  'locus-4',
  'locus-5',
  'locus-6'
  ])
```

```
set3 = set([
  'locus-10',
  'locus-11',
  'locus-12'
  ])
```

membership

is an element in a set?
(very fast)

```
'locus-1' in set1
True
'locus-1' in set2
False
```

isdisjoint

do sets share any common element?

```
set1.isdisjoint(set2)
False
set1.isdisjoint(set3)
True
```

issubset

is set on left a subset of set on right?

```
set1.issubset(set2)
False
set2.issubset(set1)
True
```

issuperset

is set on left a superset of set on right?

```
set1.issuperset(set2)
True
set2.issuperset(set1)
False
```

union

new set with all elements of others

```
set2.union(set3)
{'locus-10',
 'locus-11',
 'locus-12',
 'locus-4',
 'locus-5',
 'locus-6'}
```

intersection

new set overlapping elements of others

```
set1.intersection(set2)
{'locus-4',
 'locus-5',
 'locus-6'}
```

difference

new set of non-overlapping elements of others

```
set1.difference(set2)
{'locus-1',
 'locus-2',
 'locus-3'}
```

set() vs. frozenset()

set() is mutable

```
In: set1 = set([1,2,3])
In: set1.add(4)
In: print(set1)
{1, 2, 3, 4}
```

frozenset() is not mutable

```
In: set1 = frozenset([1,2,3])
In: set1.add(4)
AttributeError: 'frozenset' object has no attribute 'update'
```

os and os.path module

"portable" interaction with OS functions

os

top-level module

methods/attrs for interacting with OS

```
os.environ
os.getenv()
os.getcwd()
os.chdir()
os.uname()
```

os.path

submodule of os

methods/attrs for interacting with paths

```
os.path.abspath()
os.path.basename()
os.path.dirname()
os.path.join()
os.path.split()
os.path.splitext()
```

os module

"portable" interaction with OS functions

os.environ

returns a dict of all environment variables

```
In:  os.environ["HOME"]
Out: '/Users/bcf'
```

os.getenv()

returns a single env variable

```
In:  os.environ["HOME"]
Out: '/Users/bcf'
```

os.getcwd()

get current working directory

```
In:  os.getcwd()
Out: '/Users/bcf/tmp'
```

os.chdir()

change to a directory

```
In:  os.getcwd()
Out: '/Users/bcf/tmp'
In:  os.chdir('/Users/bcf/Dropbox/')
In:  os.getcwd()
Out: '/Users/bcf/Dropbox'
In:  os.chdir(os.getenv("HOME"))
In:  os.getcwd()
Out: '/Users/bcf'
```

os.uname()

return specific OS information (version, type, etc.)

```
In:  os.getcwd()
Out: posix.uname_result(
sysname='Darwin',
nodename='brant-4.lsu.edu',
release='14.5.0',
version='Darwin Kernel Version 14.5.0'
...
)
```

os.path submodule

os.path.abspath()

resolves relative links

```
In: os.path.abspath("../bcf/tmp")
Out: '/Users/bcf/tmp'
```

os.path.join()

joins paths/filenames

```
In:  os.path.join('/tmp', 'dir')
Out: '/tmp/dir'
In:  os.path.join('/tmp', 'dir', 'file.txt')
Out: '/tmp/dir/file.txt'
```

os.path.split()

splits a path into head and tail portions

```
In:  os.path.split("/tmp/dir/file1.py")
Out: ('/tmp/dir', 'file1.py')
In:  os.path.split("/tmp/dir")
Out: ('/tmp', 'dir')
```

os.path.basename()

.split() shortcut; return filename

```
In:  os.path.basename("/tmp/dir/file.txt")
Out: 'file.txt'
In:  os.path.split("file.txt")
Out: ('', 'file.txt')
```

os.path.dirname()

.split() shortcut; return dirname

```
In:  os.path.dirname("/tmp/dir/file.txt")
Out: '/tmp/dir'
In:  os.path.dirname("file.txt")
Out: ''
```

os.path submodule

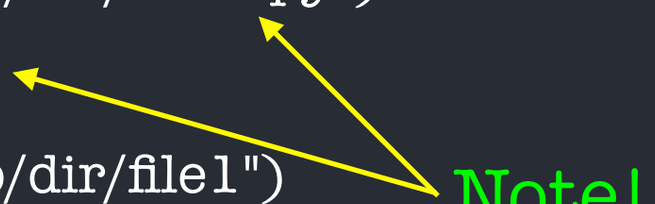
os.path.splitext()

splits a path into (root, ext) where ext is the file extension

```
In:   os.path.splitext("/tmp/dir/file1.py")
Out: ('/tmp/dir/file1', '.py')
```

```
In:   os.path.splitext("/tmp/dir/file1..py")
Out: ('/tmp/dir/file1.', '.py')

In:   os.path.splitext("/tmp/dir/file1")
Out: ('/tmp/dir/file1', '')
```



sys module

attributes or methods of the Python interpreter

sys.exit()

exit from Python
(stop execution right NOW)

sys.path

returns the search path
for Python modules ($PYTHONPATH)

```
In:  sys.path
Out: [
 '/Users/bcf/anaconda/envs/py35/bin',
 '/Users/bcf/git/phyluce',
 '/Users/bcf/git/seqtools',
 '/Users/bcf/git/splitaake',
...
]
```

sys.platform

another way to return OS type

```
In:  sys.platform
Out: 'darwin'
```

sys.prefix

site-specific directory prefix
where the platform independent
Python files are installed

```
In:  sys.prefix
Out: '/Users/bcf/anaconda/envs/py35'
```

sys.setrecursionlimit

set the maximum depth of the
Python interpreter stack. The highest
possible limit is platform-dependent.

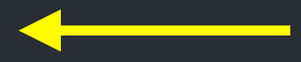
```
In:  sys.getrecursionlimit()
Out: 1000
In:  sys.setrecursionlimit(10000)
In:  sys.getrecursionlimit()
Out: 10000
```

copy module

```
In:   my_list = [1, 2, 3, 4]
In:   my_other_list = my_list
In:   print(my_other_list)
Out: [1, 2, 3, 4]
```

Expected

```
In:   my_list[1] = 'gremlins'
Out: [1, 'gremlins', 3, 4]
```



Expected

```
In:   print(my_other_list)
Out: [1, 'gremlins', 3, 4]
```

Whoah!!

copy module

```
In:   my_list = [1, 2, 3, 4]
In:   my_other_list = my_list
In:   print(my_other_list)
Out: [1, 2, 3, 4]

In:   my_list[1] = 'gremlins'
Out: [1, 'gremlins', 3, 4]

In:   print(my_other_list)
Out: [1, 'gremlins', 3, 4]
```

Let's look at memory addresses of each variable

```
In:   hex(id(my_list))
Out: 0x1036c6c88
```

```
In:   hex(id(my_other_list))
Out: 0x1036c6c88
```

copy module

Let's look at memory addresses of each variable

```
In:  hex(id(my_list))
Out: 0x1036c6c88
```

```
In:  hex(id(my_other_list))
Out: 0x1036c6c88
```

They are identical

Variable names point to same address in memory

So how do we make a real copy?

copy module

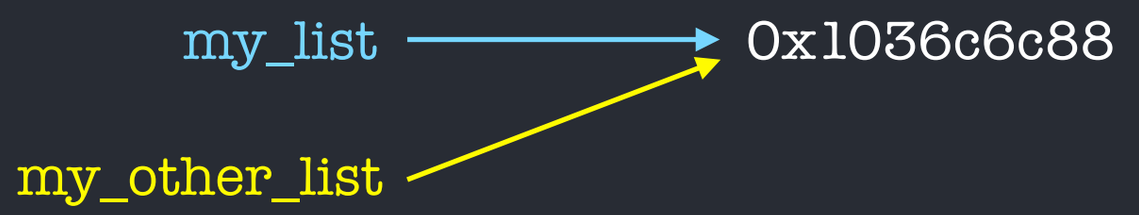
Let's look at memory addresses of each variable

```
In:  hex(id(my_list))
Out: 0x1036c6c88
```

```
In:  hex(id(my_other_list))
Out: 0x1036c6c88
```

They are identical

Variable names point to same address in memory



So how do we make a real copy of an object?

copy module

```
In:   import copy
In:   my_list = [1, 2, 3, 4]
In:   my_other_list = copy.copy(my_list)
In:   hex(id(my_list))
Out: 0x100571788
In:   hex(id(my_other_list))
Out: 0x1036d7c88
```

```
In:   my_list[1] = 'gremlins'
In:   print(my_list)
Out: [1, 'gremlins', 3, 4]
In:   print(my_other_list)
Out: [1, 2, 3, 4]
```

Expected

Expected

copy.copy() returns a "shallow" copy of my_list

but, what's a "shallow" copy of my_list ?

AKA, it's *not* so simple…

copy module

```
In:   import copy
In:   my_list = [1, ['gremlins', 'are', 'bad'], 3, 4]
In:   my_other_list = copy.copy(my_list)
In:   hex(id(my_list))
Out: 0x1036231c8
In:   hex(id(my_other_list))
Out: 0x1036d7988


In:   my_list[1][0] = 'cats'
In:   print(my_list)
Out: [1, ['cats', 'are', 'bad'], 3, 4]
In:   print(my_other_list)
Out: [1, ['cats', 'are', 'bad'], 3, 4]
```

Whoah. How?

copy module

```
In:   import copy
In:   my_list = [1, ['gremlins', 'are', 'bad'], 3, 4]
In:   my_other_list = copy.copy(my_list)
In:   hex(id(my_list))
Out: 0x1036231c8
In:   hex(id(my_other_list))
Out: 0x1036d7988


In:   my_list[1][0] = 'cats'
In:   print(my_list)
Out: [1, ['cats', 'are', 'bad'], 3, 4]
In:   print(my_other_list)
Out: [1, ['cats', 'are', 'bad'], 3, 4]
```

Whoah. How?

because copy.copy() returns a "shallow" copy of my_list

```
In:   hex(id(my_list[1]))
Out: 0x1036d79c8
```

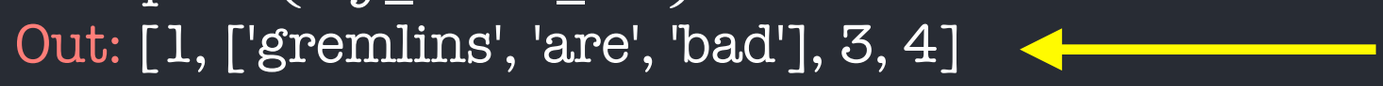
```
In:   hex(id(my_other_list[1]))
Out: 0x1036d79c8
```

They are identical!

copy module

```
In:   import copy
In:   my_list = [1, ['gremlins', 'are', 'bad'], 3, 4]
In:   my_other_list = copy.deepcopy(my_list)
In:   hex(id(my_list))
Out: 0x1036cfc08
In:   hex(id(my_other_list))
Out: 0x1036d7688

In:   my_list[1][0] = 'cats'
In:   print(my_list)
Out: [1, ['cats', 'are', 'bad'], 3, 4]
In:   print(my_other_list)
Out: [1, ['gremlins', 'are', 'bad'], 3, 4]
```



Expected

copy.copy() returns a "shallow" copy of my_list - doesn't worry about nested objects

copy.deepcopy() returns a "deep" copy of my_list
it recursively copies all objects and items in objects

```
In:   hex(id(my_list[1]))
Out: 0x1036d7e48
```

```
In:   hex(id(my_other_list[1]))
Out: 0x1036d7908
```

They are different!

Dealing with lots of files...

we have some files in /tmp/my_files:

```
file1.doc   file1.gif   file2.doc   file2.gif   file3.doc   file3.gif   file4.doc   file4.gif   file5.doc   file5.gif
file1.fasta file1.tiff  file2.fasta file2.tiff  file3.fasta file3.tiff  file4.fasta file4.tiff  file5.fasta file5.tiff
file1.fastq file1.txt   file2.fastq file2.txt   file3.fastq file3.txt   file4.fastq file4.txt   file5.fastq file5.txt
```

How can get get only those .fastq files?

glob module

Finds all pathnames matching a specified pattern

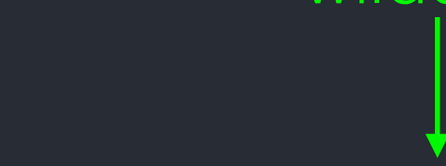
(according to Unix shell expansion rules)

we have some files in /tmp/my_files:

```
file1.doc   file1.gif   file2.doc   file2.gif   file3.doc   file3.gif   file4.doc   file4.gif   file5.doc   file5.gif
file1.fasta file1.tiff  file2.fasta file2.tiff  file3.fasta file3.tiff  file4.fasta file4.tiff  file5.fasta file5.tiff
file1.fastq file1.txt   file2.fastq file2.txt   file3.fastq file3.txt   file4.fastq file4.txt   file5.fastq file5.txt
```

```
In:   import os
In:   import glob
In:   os.chdir("/tmp/my_files")
```

wildcard



```
In:   glob.glob('*.fastq')
Out:  ['file1.fastq', 'file2.fastq', 'file3.fastq', 'file4.fastq', 'file5.fastq']
```

glob module

Finds all pathnames matching a specified pattern

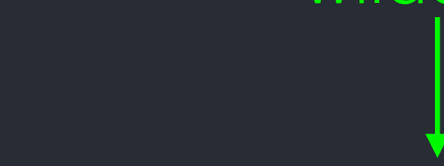
(according to Unix shell expansion rules)

we have some files in /tmp/my_files:

```
file1.doc   file1.gif   file2.doc   file2.gif   file3.doc   file3.gif   file4.doc   file4.gif   file5.doc   file5.gif
file1.fasta file1.tiff  file2.fasta file2.tiff  file3.fasta file3.tiff  file4.fasta file4.tiff  file5.fasta file5.tiff
file1.fastq file1.txt   file2.fastq file2.txt   file3.fastq file3.txt   file4.fastq file4.txt   file5.fastq file5.txt
```

```
In:   import os
In:   import glob
In:   os.chdir("/tmp/my_files")
```

wildcard



```
In:   glob.glob('*.fastq')
Out:  ['file1.fastq', 'file2.fastq', 'file3.fastq', 'file4.fastq', 'file5.fastq']
```

How would we get all .fastq and all .fasta files?

glob module

Finds all pathnames matching a specified pattern

(according to Unix shell expansion rules)

we have some files in /tmp/my_files:

```
file1.doc   file1.gif   file2.doc   file2.gif   file3.doc   file3.gif   file4.doc   file4.gif   file5.doc   file5.gif
file1.fasta file1.tiff  file2.fasta file2.tiff  file3.fasta file3.tiff  file4.fasta file4.tiff  file5.fasta file5.tiff
file1.fastq file1.txt   file2.fastq file2.txt   file3.fastq file3.txt   file4.fastq file4.txt   file5.fastq file5.txt
```

```
In:   import os
In:   import glob
In:   os.chdir("/tmp/my_files")
```

```
In:   glob.glob('*.fast*')
Out:  [
'file1.fasta', 'file1.fastq',
'file2.fasta', 'file2.fastq',
'file3.fasta', 'file3.fastq',
'file4.fasta', 'file4.fastq',
'file5.fasta', 'file5.fastq'
]
```

glob module

Finds all pathnames matching a specified pattern
(according to Unix shell expansion rules)

we have some files in /tmp/my_files:

```
file1.doc   file1.gif   file2.doc   file2.gif   file3.doc   file3.gif   file4.doc   file4.gif   file5.doc   file5.gif
file1.fasta file1.tiff  file2.fasta file2.tiff  file3.fasta file3.tiff  file4.fasta file4.tiff  file5.fasta file5.tiff
file1.fastq file1.txt   file2.fastq file2.txt   file3.fastq file3.txt   file4.fastq file4.txt   file5.fastq file5.txt
```

When we use full path in glob,
we get full path in results

```
In:   import glob
In:   glob.glob('/tmp/my_files/*.fast*')
Out:  [
'/tmp/my_files/file1.fasta', '/tmp/my_files/file1.fastq',
'/tmp/my_files/file2.fasta', '/tmp/my_files/file2.fastq',
'/tmp/my_files/file3.fasta', '/tmp/my_files/file3.fastq',
'/tmp/my_files/file4.fasta', '/tmp/my_files/file4.fastq',
'/tmp/my_files/file5.fasta', '/tmp/my_files/file5.fastq'
]
```

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

map()

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

example14.py

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

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

(here we make power a variable)

File 0 | Project 0 | No Issues | example14.py | 13:27 | LF | UTF-8 | Python

map()

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

example14.py

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

Here we do same… but we pass a 2nd iterable holding values for power, which are applied in parallel to function

If we do this, 2nd iterable must be as long as first

File 0 Project 0 No Issues example14.py 10:45 LF UTF-8 Python

itertools module

time and datetime modules

Collections module