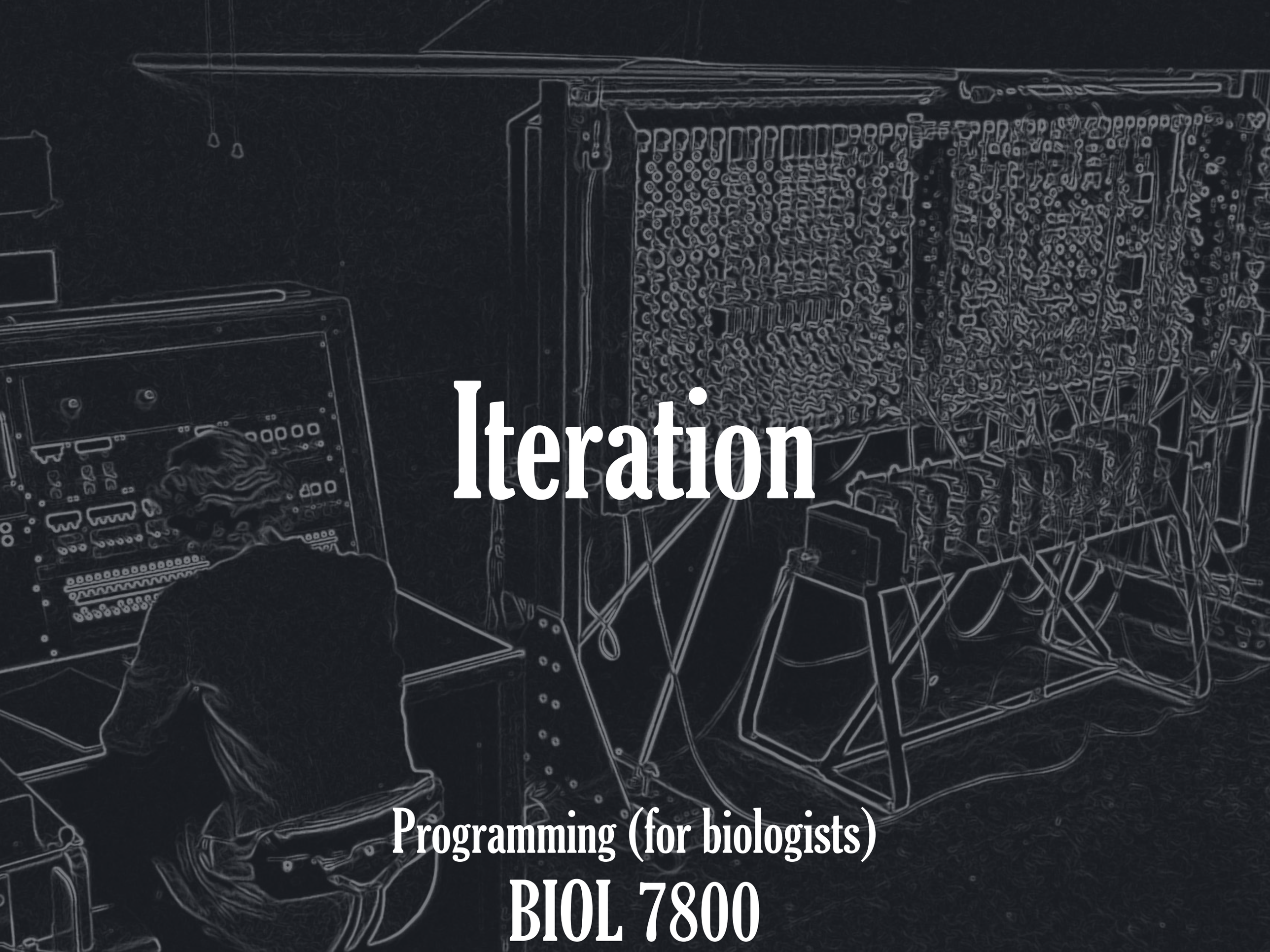

Iteration

Programming (for biologists)

BIOL 7800

Functions

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

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4   """
5   example.py
6
7   Created by Brant Faircloth on 3 February 2016
8   Copyright 2016 Brant C. Faircloth.  All rights reserved.
9   """
10
11
12  def function1(arg):
13      product = arg * arg
14      return product
15
16
17  def main():
18      function1(2)
19
20  if __name__ == '__main__':
21      main()
22
```

File 0 | Project 0 | No Issues | example.py* | 18:16 | LF | UTF-8 | Python

Functions allows us to divide our programs into atomic parts that are easier to **understand**, **debug**, and **use again**.

What is computer programming?

Computer programming
(often shortened to `*programming*`)
is a process that leads from an original formulation of a computing problem to executable computer programs.

- Wikipedia

Providing a logical sequence of arguments to a computer so that it can perform a desired task

- BIOL7800

Ada Lovelace
(1815 - 1852)

Many of our desired tasks are repetitive.

We have 1000 numbers, and we want to
do something with each of them

example7.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example7.py

```
1   # !/usr/bin/env python
2   # encoding: utf-8
3
4   def function_to_do_something(numbers):
5       #pseudocode
6       for number in numbers:
7           do_something_with_each_number
8
9
10  def main():
11      numbers = range(0, 1000)
12
13  if __name__ == '__main__':
14      main()
15
```

Many of our desired tasks are repetitive.

1. gunzip -c AR001_CATTCCT_L008_R1_001.fastq.gz | less -S (ssh)

```
@3KP75M1:611:C3NBEACXX:8:1101:2032:2202 1:N:0:CATTCCT
GGCACCACCTTCAGTGCAGAGCTCAACGGATCCTCCAAGCCAAGGATGGAGATCACTGGTTTCGGGAGATTTGGACCCAAATCTGGTAGATCGGAAGAGC
+
CCCFFFFFHHHHHJIJJJJJ3FHJJJJJJJJJJJJJJJJJIJJIIJJJJJJJJJJJJJJJJIJJHHHFFDCEDDDDDDDDDDDDDDEDDDDDEDDDDDDDDC
@3KP75M1:611:C3NBEACXX:8:1101:2239:2240 1:N:0:CATTCCT
TCCTCAACAGCATGAGTCCTGTGTGGCCTTGCATTGTCATCCATAAAAACAAAGTCGGGACCAATGGCACCCCGGAAAAGACGCAAGTGGAGGAGGATAA
+
CCCFFFFFHHHHHJJJJJJJJIJJJJJJJJJJJJJJJGIJJJJJJIJJJJJJJJJHIJJJIIJJHHHHHFFFFDDDBDDDDDDDDDDCCDDDDDDDBCDD
@3KP75M1:611:C3NBEACXX:8:1101:2438:2029 1:N:0:CATTCCT
TGTTNATTCTATGAGGTAGTTTTGTTTAAATCTTGATTACATTAATTGGAAACTAAATCGCTTCTCTCTTTAGCAGTATCGGATGTGTGTGTGTTCAGGA
+
CCCF#2ADHHHHHJJJFHIHJJJJJJJIJJJJJJJIJJJJJJJJJJJJJJJJJJJIIJIJJJJJJJJGIJJIIJIHHFHHHFFFDEEFDDDDDDCCDDDC
@3KP75M1:611:C3NBEACXX:8:1101:2335:2127 1:N:0:CATTCCT
GTGTTAATAAACTAAATCTGTAGGTTGGATGCAATGAAAGGGTAGAATCGTGAGGAATTCTATGGACTAGACTTCCCATAGAGACGTTTTGAATGGTATC
+
B@@FDFFFHHHHHIJJJJJJJJJIIIIJHHGIJJJJJJIJJJ;FFHIJJJHIIGHEIIJGHGJJICHIIGIIGIGGHHFD>CDEFCCDEBBBDCDCD@CA
@3KP75M1:611:C3NBEACXX:8:1101:2411:2147 1:N:0:CATTCCT
ATAGGTAAATTATAAAAATGTTTATATTTTTCAGTAAGAAAGTTTCAGTTGCGGCCGGAAATAATCAAGAATGAATTCTAAGTAATTTGGTACTGAGAAA
+
CCCFFDEFHHHHHJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJJHIJJIJJJJJJJJJJJ
@3KP75M1:611:C3NBEACXX:8:1101:2316:2149 1:N:0:CATTCCT
AAAGGAAGAAATAATAATAACCATATGGATAGAAAAAACAATTGTTGTTGTTGAGTGC
+
CCCFFFFFHHHHHJJJJJJJJJJJJJJJJIJJJJJJJJJJJJJIJJJJJJJJJJJJIJHHJ
@3KP75M1:611:C3NBEACXX:8:1101:2282:2149 1:N:0:CATTCCT
ATCAATGCGTCCTACCAGTCCTTTCATAAGGATCTGAGGCATGGACATTAACTAAGAA
+
??=AADA?FCFFFGHBGIBHIIGIIIICEHG??FHB?FGCBGDDGCFBBBDFGFD@G>
@3KP75M1:611:C3NBEACXX:8:1101:2513:2168 1:N:0:CATTCCT
TATTTCACATCTGGCTCCTATTTTTGACTTATTTTCCTCCTCATTAACTTTTCCTCGT
+
CCCFFFFFGHHHHJJJJJJJJJJJJIIJJJJJJJJJGGHIGEHEHIIIEHIJIIGEHGI
:
```

We have 30 M sequences, and we want to do something with **each** of them

example7.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example7.py

```
1  # !/usr/bin/env python
2  # encoding: utf-8
3
4  def function_to_do_something(numbers):
5      #pseudocode
6      for sequence in list_of_sequences:
7          do_something_with_each_sequence
8
9
10 def main():
11     sequences = list_of_sequences
12
13 if __name__ == '__main__':
14     main()
15
```

Iteration (AKA “loops”)

iteration |ˌitəˈrāSHən|
noun
the repetition of a process or utterance.

• repetition of a mathematical or computational procedure applied to the result of a previous application, typically as a means of obtaining successively closer approximations to the solution of a problem.

ORIGIN late Middle English: from Latin ***iteratio(n-)***, from the verb ***iterare*** (see **iterate**) .

Provides the ability to **run a statement** or a **block of statements** repeatedly

(i.e., this includes functions)

http://www.whereswaldo.com/m/images/maps_troy.jpg

Iteration

(you've seen it before - but where?)

Iteration

(by recursion)

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

andre_recursion.py

```
"""
My recursive summing program
Created by Andre Moncrieff on 2 Feb 2016.
Copyright 2016 Andre E. Moncrieff. All rights reserved.
"""
import sys
sys.setrecursionlimit(1100)


def recursion_sum(x):
    """
    Sums each integer from x down through zero.
    This function is based on solution #2 on the following page:
    http://www.python-course.eu/python3_recursive_functions.php

    """
    if x == 0:
        return 0
    else:
        return x + recursion_sum(x-1)


def main():
    answer = recursion_sum(1000)
    print(answer)

if __name__ == '__main__':
    main()
```

File 0 Project 0 ✔ No Issues andre_recursion.py* 32:11 LF UTF-8 Python

Iteration

(by recursion)

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

andre_recursion.py

```
"""
My recursive summing program
Created by Andre Moncrieff on 2 Feb 2016.
Copyright 2016 Andre E. Moncrieff. All rights reserved.
"""
import sys
sys.setrecursionlimit(1100)


def recursion_sum(x):
    """
    Sums each integer from x down through zero.
    This function is based on solution #2 on the following page:
    http://www.python-course.eu/python3_recursive_functions.php

    """
    if x == 0:
        return 0
    else:
        return x + recursion_sum(x-1)


def main():
    answer = recursion_sum(1000)
    print(answer)

if __name__ == '__main__':
    main()
```

File 0 Project 0 ✔ No Issues andre_recursion.py* 32:11 LF UTF-8 Python

(1000 + recursion_sum(999))

(999 + recursion_sum(998))

(998 + recursion_sum(997))

(997 + recursion_sum(996))

(996 + recursion_sum(995))

...

(1 + recursion_sum(0))

(0)

+

~ 500,000

Iteration

3 Major Flavors

recursion

(somewhat rare)

```
def func1(x=1000):
    if x == 0:
        return 0
    else:
        return x + func1(x-1)
```

while

(rare)

```
def func1(x=0):
    i = 0
    while i < 1000:
        i += 1
        x += i
    return x
```

for

(common)

```
def func1(x=0):
    for i in range(0,1001):
        x += i
    return x
```

Iteration

3 Major Flavors

Focus on these

recursion

(somewhat rare)

```
def func1(x=1000):
    if x == 0:
        return 0
    else:
        return x + func1(x-1)
```

while

(rare)

```
def func1(x=0):
    i = 0
    while i < 1000:
        i += 1
        x += i
    return x
```

for

(common)

```
def func1(x=0):
    for i in range(0,1001):
        x += i
    return x
```

Iteration

`while`

A **`while`** loop is (usually) conditional, running the statements in the loop until it's condition is met

Anatomy of a **while**

A **while** loop is (usually) conditional, running the statements in the loop until it's condition is met

while statement

the "condition"

```
while <condition>:
    <the thing to do repeatedly>
```

what we want to keep doing,
if the **condition** is met

Anatomy of a **while**

A **while** loop is (usually) conditional, running the statements in the loop until it's condition is met

while statement

the "condition"

```
x= 0
while x < 100:
    x += 2
    print(x)
```

} This will run continuously until `x >= 100`

what we want to keep doing,
if the **condition** is met

Anatomy of a while

What's wrong here?

while statement

the "condition"

```
x= 0
while x != 'a':
    x += 2
    print(x)
```

} This will run continuously

what we want to keep doing,
if the **condition** is met

Anatomy of a while

What's wrong here?

while statement

the "condition"

```
x= 0
while x != 'a':
    x += 2
    print(x)
```

This will run ~~continuously~~ infinitely ^

what we want to keep doing,
if the **condition** is met

Anatomy of a while

breaking out of a **while** loop

while statement

the "condition"

```
x= 0
while x < 100:
    if x == 49:
        break
    else:
        x += 2
        print(x)
```

But if x == 49, we'll break out of the loop and return to the program

This will run continuously until x >= 100

Anatomy of a **while**

while can be used to keep a program running (forever)

while statement

the “condition”

```
while True:
    <the thing to do repeatedly>
```

what we want to keep doing,
if the **condition** is met

Anatomy of a while

while can be used to keep a program running (forever)

```
while True:
    response = input('> What is the velocity of an ' +
                     'unladen swallow?')
    if 13.0 <= float(response) <= 14.0:
        break
    else:
        print(response)

print('Correct!')
```

Iteration

3 Major Flavors

Focus on these

recursion

(somewhat rare)

```
def func1(x=1000):
    if x == 0:
        return 0
    else:
        return x + func1(x-1)
```

while

(rare)

```
def func1(x=0):
    i = 0
    while i < 1000:
        i += 1
        x += i
    return x
```

for

(common)

```
def func1(x=0):
    for i in range(0,1001):
        x += i
    return x
```

Iteration

for

A **for** loop traverses the values in an **iterator / iterable**,
running the statements in the loop across all values

So, what's an
iterator / iterable ?

A variable (or object) that we can iterate over

AKA lots of things

(lists, strings, tuples, dictionaries, files, lines, etc.)

Anatomy of a for

for statement

the iterator

```
for item in <iterator>:
    <the thing to do with each item>
```

the thing we wish
to do with **each item**

Anatomy of a for

for statement

the iterator

```
for item in 'doggie':
    print(item)
```

the thing we wish
to do with **each item**

Anatomy of a for

for statement

the iterator

```
for n in [1, 2, 3, 4]:
    print(n)
```

the thing we wish
to do with **each item**

Anatomy of a for

breaking out of a **for** loop

for statement

the iterator

```
for n in [1, 2, 3, 4]:
    if n == 2:
        break
else:
    print(n)
```

the thing we wish
to do with **each item**

Anatomy of a for

ending a **for** loop

When does a **for** loop finish?

for statement

execution of loop ends at end of iterator

```
for n in [1, 2, 3, 4]:
    print(n)
```