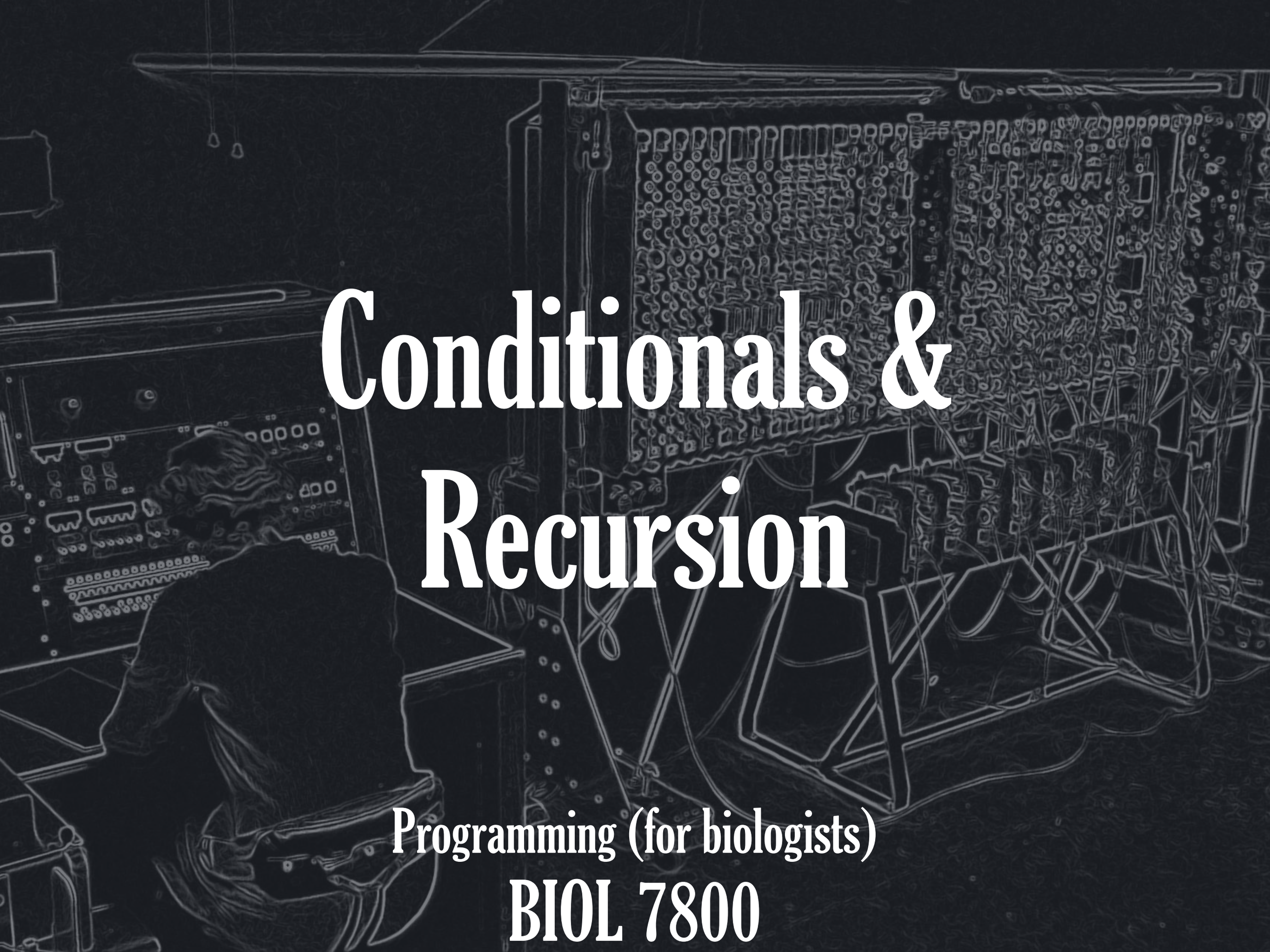

Conditionals & Recursion

Programming (for biologists)

BIOL 7800

Functions

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

example.py

```python
#!/usr/bin/env python
# encoding: utf-8

"""
example.py

Created by Brant Faircloth on 27 Jan 2016.
Copyright 2016 Brant C. Faircloth. All rights reserved.
"""


def function1():
    print("BIOL 7800")


def main():
    function1()

if __name__ == '__main__':
    main()

```

File 0 Project 0 No Issues example.py* 7:42 LF UTF-8 Python

Anatomy of a Function

the function **name**

example.py — /Users/bcf/Dropbox (faircloth-lab)/Clas

example.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3
4
5   def fname(arg):
6       """function description"""
7       # this is an empty function
8       pass
9
```

def

an **argument** (optional)
(passed to function)

a description
(can access)
fname.__doc__

basically means skip
fxn for now.
(this is where your code will go)

Anatomy of a Function

Arguments can also have **default** values that can be overridden

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

example.py

```python
#!/usr/bin/env python
# encoding: utf-8


def fname(arg1="rabbits"):
    """function description"""
    # this is an empty function
    print(arg1)
    print("are")
    print("better")


fname("scorpions")
```

We’re overriding the default

File 0 | Project 0 | No Issues | example.py* | 1:1 | LF | UTF-8 | Python

Anatomy of a Function

Functions can **return** values (but don't **have** to)

example.py — /Users/bcf/Dropbox (fair

example.py

```python
#!/usr/bin/env python
# encoding: utf-8


def fname(arg1):
    """function description"""
    # this is an empty function
    print(arg1 + arg1)


fname(2)
```

File 0 Project 0 No Issues example.py* 8:23

"void" functions

(do not return a value)

example.py — /Users/bcf/Dropbox (fair

example.py

```python
#!/usr/bin/env python
# encoding: utf-8


def fname(arg1):
    """function description"""
    # this is an empty function
    result = arg1 + arg1
    return result


from_fname_function = fname(2)
```

File 0 Project 0 No Issues example.py* 6:31

"fruitful" functions

(return a value)

"global" and "local" refer to Variable Scope

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

example.py

```python
# encoding: utf-8

ARG1 = "hot dogs"

def fname():
    """function description"""
    # this is an empty function
    print(ARG1)
    print("are")
    print("better")


fname()
print(ARG1)
```

But, if you use them, the convention is to **CAP** their names

That way, you ***know*** they are global

File 1 | Project 1 | ✖ 1 Issue | example.py* | 16:1 | LF | UTF-8 | Python

Variable Scope

main() function helps ensure variables are **"encapsulated"**

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

example.py

```python
#!/usr/bin/env python
# encoding: utf-8


def fname(arg1):
    """function description"""
    # this is an empty function
    print(arg1)
    print("are")
    print("better")


def main():
    var1 = "hot dogs"
    fname(var1)

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

File 0 | Project 0 | ✔ No Issues | example.py* | 18:8 | (1, 6) | LF | UTF-8 | Python

All variables here are encapsulated, thus "local"

They can't easily **"leak"** to other, <u>unintended</u> functions

Flow of Execution

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

example.py

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 26 12:41:32 2016
@author: Marco
"""


def main():
    # I nested one function inside the other
    def get_numbers():
        # creating a list with the numbers in the desired rang
        print([elem for elem in range(0, 51, 5)])
    return get_numbers()


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

What is the flow of execution here?

File 0 | Project 0 | ✔ No Issues | example.py* | 18:1 | LF | UTF-8 | Python

Flow of Execution

What if I want to control the flow of execution?

example2.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example2.py

```
#!/usr/bin/env python
# encoding: utf-8


def function2(number):
    pass


def function1(number):
    pass


def main():
    pass

if __name__ == '__main__':
    main()

```

File 0 | Project 0 | No Issues | example2.py* | 14:9 | LF | UTF-8 | Python | 1 update

Flow of Execution

What if I want to control the flow of execution?

example2.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example2.py

```python
#!/usr/bin/env python
# encoding: utf-8


def function2(number):
    pass


def function1(number):
    pass


def main():
    # in some cases i want to run function 1
    function1(number)
    # but in other cases, i want to run function 2
    function2(number)

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

Conditionals

They alter the order of execution or the “flow” of a program

example2.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example2.py

```
#!/usr/bin/env python
# encoding: utf-8


def function2(number):
    pass


def function1(number):
    pass


def main():
    # in some cases i want to run function 1
    if something is True:
        function1(number)
    # but in other cases, i want to run function 2
    else:
        function2(number)

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

Conditionals

They alter the order of execution or the “flow” of a program

Conditionals (or conditional expressions) are statements that allow us to **check certain conditions** and *alter the flow* of a program based on the result.

Because they alter the flow of execution, conditionals are one type of **flow control statement.**

Anatomy of a conditional

The simplest conditional statement is "**if**"

if statement

the "condition"

```
if <something>:
    <the thing to do>
```

what we want to do,
if the **condition** is met

Anatomy of a conditional

if statement

the “condition”

```
if <something>:
    <the thing to do>
```

The condition is usually a **boolean** or a **boolean test**, meaning that its result evaluates to **True**/**False**

Boolean expressions

An expression that is either **True** or **False**

```
5 == 5
True
```

True and **False** are special values with a type of '**bool**'

```
type(True)
<class 'bool'>
```

```
type(False)
<class 'bool'>
```

Boolean expressions

An expression that is either **True** or **False**

Often use **relational operators**

`x == y` `x != y` `x < y` `x <= y` `x > y` `x >= y`

(there is no such relational operator as **=<** or **=>**)

Anatomy of a conditional

if statement

the “condition”

```
if x < 5:
    x = x + 1
```

what we want to do,
if the **condition** is met

Anatomy of a conditional

What if we want to test **two** conditions?

if statement

the “condition”

```
if <something> and <something else>:
    <the thing to do>
```

what we want to do,
if the **condition** is met

Anatomy of a conditional

What if we want to test **two** conditions?
We use a **"logical operator"**

logical operator

if statement

condition #1

condition #2

```
if <something> and <something else>:
    <the thing to do>
```

what we want to do,
if the **condition** is met

Logical operators

and	or	not
True only if both conditions are met	True if one or the other condition is met	Negates an expression
`x < 5 and y < 5`	`x < 5 or y < 5`	`not x < 5`

Anatomy of a conditional

logical operator

if statement

the “condition”

```
if x < 5 and y < 5:
    x = x + y
```

what we want to do,
if the **condition** is met

Logical operators

The curious case of **not**

not is often used in conjunction with the **is** keyword

```
if x is not 5:
    <do something>
```

not is also used in conjunction with the **in** keyword

```
x = 5
if x not in [1,2,3,4]:
    <do something>
```

Anatomy of a conditional

if... else...

"alternative execution"

if the boolean is met, do something; **else**, do something different

```
if x < 5 and y < 5:
    x = x + y
else:
    x = x - y
```

Do one thing

Do another thing

Anatomy of a conditional

if... else...

"alternative execution"

if the boolean is met, do something; **else**, do something different

```
if x < 5 and y < 5:
    x = x + y
else:
    x = x - y
```

Do one thing

But what if we have more conditions?

Anatomy of a conditional

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

"chained conditionals"

if the boolean is met, do something; **else**, do something different

```
if x < 5 and y < 5:
    x = x + y
elif x < 6 and y < 6:
    x = x * y
else:
    x = x - y
```

Do one thing

Do another thing

If none of those, do a third thing

Anatomy of a conditional

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

"chained conditionals"

```
if x < 5 and y < 5:
    x = x + y
elif x < 6 and y < 6:
    x = x * y
elif x < 7 and y < 7:
    x = x % y
elif x < 8 and y < 8:
    x = x // y
else:
    x = x - y
```

Can chain as many as needed...

But, (**!!**) that doesn't mean you ***should.***

Anatomy of a conditional

"nested conditionals"

```
if x < 5 and y < 5:
    do_something_1()
    if x < 2 and y < 2:
        do_something_2()
    else:
        do_something_3()
else:
    do_something_4()
```

Can nest as many levels as needed…

But, (**!!**) that doesn't mean you ***should.***

Anatomy of a conditional

Logical operators can simplify nested conditionals

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

versus

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

Anatomy of a conditional

Logical operators can simplify nested conditionals

…and parentheses can clarify meaning.

```
if (0 < x) and (x < 10):
    print('x is a positive single-digit number.')
```

Recursion

As you've seen, functions can call other functions…

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

example.py

```python
#!/usr/bin/env python
# encoding: utf-8

"""
example.py

Created by Brant Faircloth on 27 Jan 2016.
Copyright 2016 Brant C. Faircloth. All rights reserved.
"""


def function1():
    print("BIOL 7800")


def main():
    function1()

if __name__ == '__main__':
    main()

```

File 0 Project 0 ✔ No Issues example.py* 7:42 LF UTF-8 Python

Recursion

But functions can also call themselves…

example2.py — /Users/bcf/Dropbox (faircloth-lab)/Classes/BIOL7800/temp

example2.py

```python
#!/usr/bin/env python
# encoding: utf-8


def appreciate(var1):
    if var1 <= 8:
        print(str(var1) + ' ', end='')
        var1 = var1 + 2
        appreciate(var1)
    else:
        print(": Who do we appreciate?")


def main():
    appreciate(2)


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

Recursion

Recursion is powerful, but also tricky…

```
def my_function():
    my_function()
```

"Infinite recursion"

To avoid infinite recursion, you need to define some **base case** that will eventually be met (or your program will run **forever**!)

User input

We'll cover several methods of gatheing of user input during the course… the first being so called **"raw"** input

Usually, we **ask some question**, then **prompt** for input…

```
print("How much wood could a woodchuck chuck?")
wood = input()
```

User input

We can combine the two to make a cleaner interface

```
question = "How much wood could a woodchuck chuck? "
wood = input(question)
```

What do you think is the "**type**" of the response we receive?