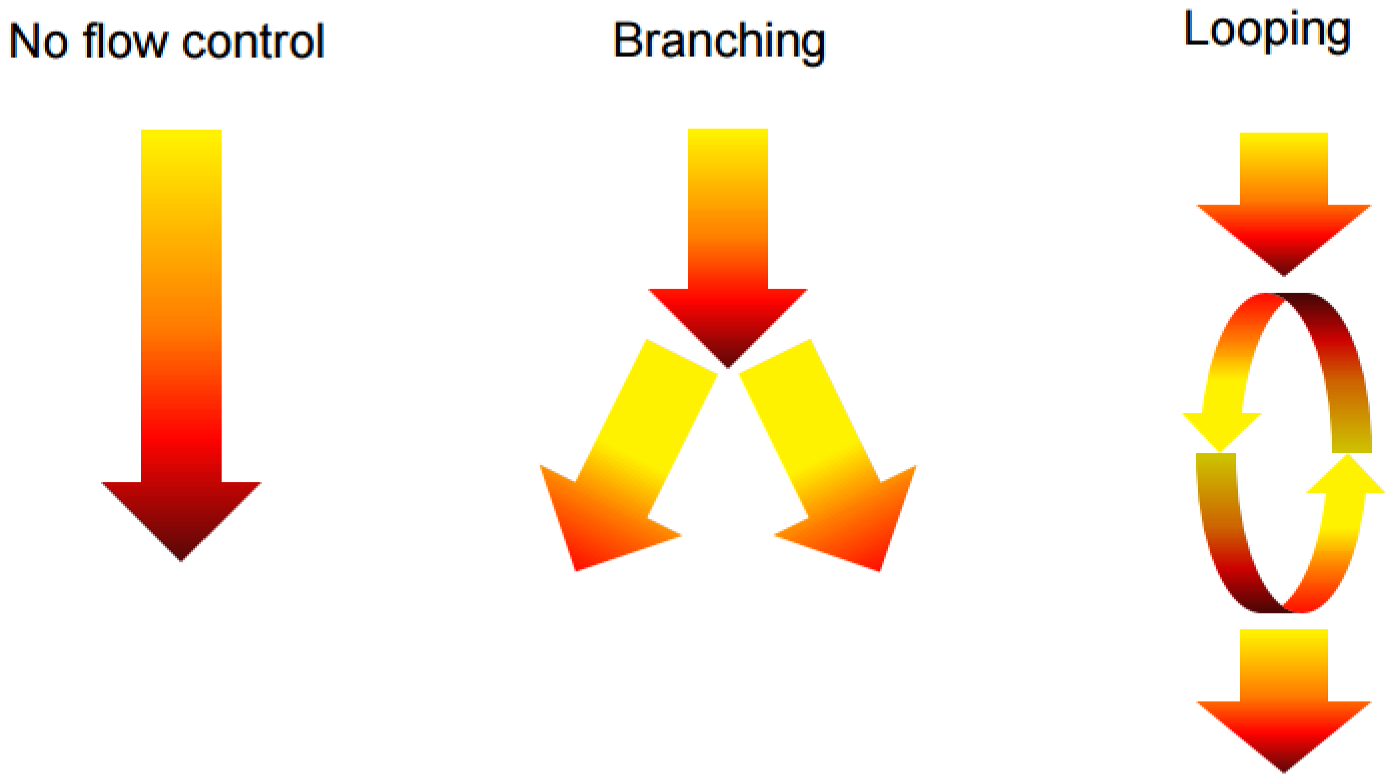
Week 3: Python Booleans and Conditionals

COURSE WEBSITE:

http://www.cs.uoregon.edu/Classes/15U/cis122

Flow Control

So far, we have only seen our programs execute in a straight line. (Procedural).



Bools!

- To allow our programs to branch we need to be able to have some kinds of tests and depending on the result we go different directions
- In order to do *that* we need a new type: the humble bool (short for boolean)
- bools can only have two values, either **True** or **False** (both keywords). Notice both **True** and **False** are capitalized.

Operators that return bools

- There are a number of operators which return bools, we call them comparison operators.

Operator	Effective Meaning
==	are these equal?
!=	are these not equal?
<>	same as != above
>	is the first 'greater' than the second?
<	is the first 'less' than the second?
>=	is the first 'greater' than or equal to the second?
<=	is the first 'less' than or equal to the second?

IF

- Great poem by Rudyard Kipling.

- The if statement gives us our first flow control mechanism. Notice if is a keyword. How does it work?

```
if 5 < 10:
        print("Five is less than ten.")
```

If: Pseudocode

```
if (some conditional):
        (code to run if the conditional is True)
```

- *This is pseudocode, not real code*
- Note the code to be run is indented, just like with the function definitions. Also note like a function definition the indent comes after a colon. The indent serves the same purpose- to group some text together.

An example of the if statement

```
def less_than_ten(x):
        if x < 10:
                print("Your number is less than ten.")
                return
        print("your number is more than ten")
```

- The second print would only happen if the conditional were **False**.
- P.S. There's a logic error in the program above. See it?

Else-A

- Haha Just kidding. Just had to let that one go.

- We may want to have a branch that specifically is followed if the if conditional is false. We could have a separate if of the opposite of our first condition but that's clunky. Instead we can use else.
- You can only have an else with an if, it cannot exist by itself. For a given if you can only have one else. Let's rework our code using else.

```
def less_than_ten(x):
if x < 10:
        print("Your number is less than ten.")
else:
        print("your number is more than ten")
```

ELSE-IF OR ELIF

- Cool. But we had that logic error when x = 10. We can fix that with the last component of **if** conditionals, else-if abbreviated as **elif**.
- Like **else**, you can only have an **elif** with an **if**, it cannot exist by itself. Unlike **else**, you can have any number of **elif**s for each **if**. Let's rework our code using **elif**.

```
def less_than_ten(x):
if x < 10:
        print("Your number is less than ten.")
elif x ==10:
         print("Your number is equal to ten.")
else:
        print("your number is more than ten")
```

How does it all work, though?

- Each test is sequential, only one branch will be followed, other two ignored. If else used then we will always follow one branch, otherwise we might follow none of them.

```
def less_than_ten(x):
    if x < 10:
        print("Your number is less than ten.")
    elif x ==10:
         print("Your number is equal to ten.")
    else:
        print("your number is more than ten")
```

THAT WILL BE ALL FOR TODAY

Psych! Group Question time:

Remember the calculator we built? Can we make it more like an actual calculator?

Welcome back!

Where we are at

Types	Functions	Flow Control	Keywords
int	print()	branching	def
float	type()		return
string	help()		None
bool	min()/max()		import
	int()/float()		if/elif/else
	round()		True/False
	len()		
	Input()		
	<user defined>		
	<turtle functions>		

A quick Test: Which of the two will work correctly?

Option 1:

```
def letter_grade(number):
        if number >= 90:
                out = 'A'
        elif number >= 80:
                out = 'B'
        elif number >= 70:
                out = 'C'
        elif number >= 60:
                out = 'D'
        else:
                out = 'F'
        print(out)
        return out
```

Option 2:

```
def letter_grade(number):
        if number >= 90:
                out = 'A'
        if number >= 80:
                out = 'B'
        if number >= 70:
                out = 'C'
        if number >= 60:
                out = 'D'
        else:
                out = 'F'
        print(out)
        return out
```

Hint: Try the Visualizer

If-elif-else v/s multiple ifs

- From the previous example we see the two ways of approaching a problem.
- But which is right?
- This is firmly dependent on the problem at hand.
- Well, ok but what does that mean?

Design decisions

- The key questions to ask yourself are these:
- should my function only ever take on branch?
  - if yes then use **if** and **elif**s
  - if no then use repeated **if**s

(think of non-exclusive traits for the latter case, like biology classifications)

- should my function always take one branch?
  - if yes, use an **else**
  - if no, don't

(if checking for some specific state or case you often don't need an **else** for example)

Practice time!

- Let us assume we are writing a function that returns whether a number is even or not.
- Lets call this function iseven()
- Given that the function archetype is iseven(num)

What would be the obvious way

```
def iseven(num):
        if num % 2 == 0:
                return True
        else:
                return False
```

Less obvious and better way

```
def iseven(num):
        return num % 2 == 0
```

That popping sound was your mind being blown.

Order of operations

order of operations:

()	parens,	
**	exponents,	
+ -	unary +/-,	CAUTION HERE
* /	multiplications,	
%	modulus,	
+ -	addition,	
==	comparison operations,	
!=		
>=		
<=		
>		
<		
and	bool operators	
or		
not		

Nested if

- You can freely nest if statements. The code below looks first to establish letter grade then has a nested if to establish any +'s or -'s. This could be done with one big if-elif-else statement but this way has some advantages

```
def letter_grade(number):
        if number >= 90:
                if number >= 98:
                        out = 'A+'
                elif number <= 92:
                        out = 'A-'
                else:
                        out = 'A'
        elif number >= 80:
                ...
```

Question Time!

- Write a function that takes a number input and outputs all the prime divisors up to 13 and denotes it prime if there are none.

Welcome back!

Bools revisited

- **bool**s are a fundamental type in python. They only have two possible values, **True** or **False**.
- **bool**s have their own operators, similar to +, -, *, etc for **int**s/**float**s.
- **bool** operators are **and**, **or**, and **not**

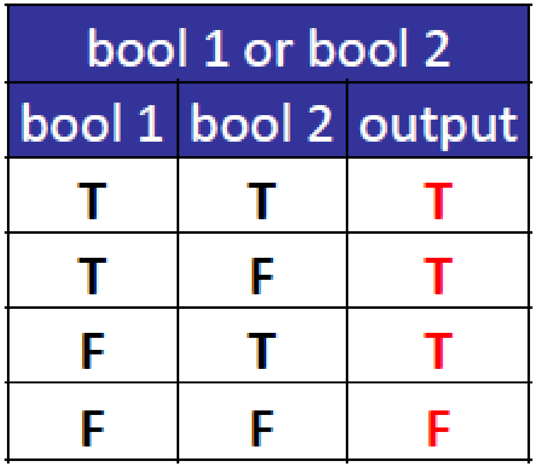
The Bool Operators: AND

- **and** is an operator on **bool**s, it returns **True** if and only if both **bool**s are **True** otherwise it returns **False**

bool 1 and bool 2		
bool 1	bool 2	output
T	T	T
T	F	F
F	T	F
F	F	F

The Bool Operators: OR

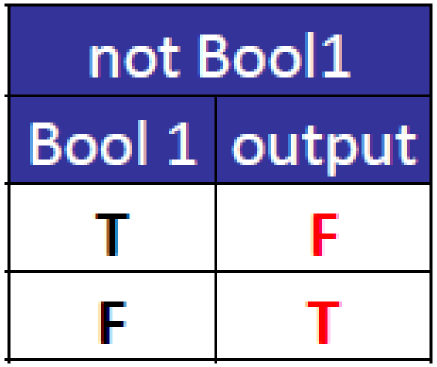
- or is the other boolean operator. It returns True if *either* of the inputs is True and False only if *both* are False.
- Think of it as the opposite of and in a way



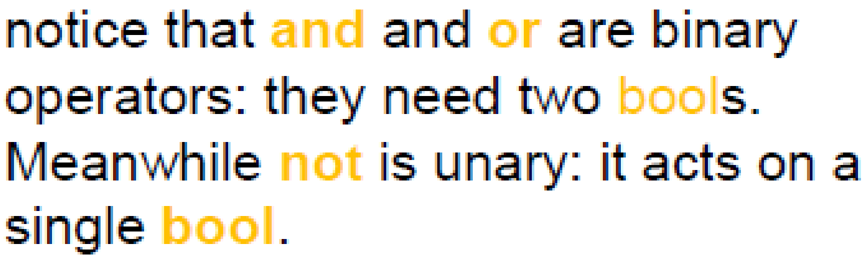
bool 1 or bool 2		
bool 1	bool 2	output
T	T	T
T	F	T
F	T	T
F	F	F

The Bool Operators: NOT

- Not is also known as the invertor. It converts True to False and False to True. Denoted by !.



not Bool1	
Bool 1	output
T	F
F	T



notice that and and or are binary operators: they need two bools. Meanwhile not is unary: it acts on a single bool.

A special case: XOR

- Xor or Exclusive or is not a boolean operator in Python. Although it is a logical operator, you will be implementing it in your project so it worth a quick look.

- XOR returns True *if and only if* exactly one of the inputs is true.

bool 1 or bool 2		
bool 1	bool 2	output
T	T	T
T	F	T
F	T	T
F	F	F

bool 1 xor bool 2		
bool 1	bool 2	output
T	T	F
T	F	T
F	T	T
F	F	F

Chaining Operators

- True and True and True and False
- Not really that different than 1 + 1 + 1 + 0

1 + (1 + (1 + 0))

1 + (1 + (1)) 1 + (1 + 1)

1 + (2)

1 + 2

3

True and (True and (True and False))
True and (True and (False))
True and (True and False)
True and (False)
True and False
False

could we have started at the other end?

IF WASN'T COMPLICATED BEFORE CHECK THIS OUT

```
True and (True and (not True or False) or (not False) and True)

True and (True and (False or False) or (True) and True)

True and (True and (False) or True)

True and (False or True)

True and (True)

True
```

not comes before and
which comes before or

Let's get weird(er)!

Try this:

```
x=1
y=2

print(x and y)
print(x or y)
```

So what happened there?

```
x = 1
y = 2
print( x and y)
print( x or y)
```

- Technically python returns one of the operands with **and** and **or**. Why? no idea. It's a design choice of the language. Essentially **or** returns the first "truthy" value it finds while **and** returns the first "falsy" value it finds. In either case if they don't find what they are looking for they return the last value they found.
- If using **True**/**False** this isn't a problem but if you start using logical operators on numbers/strings it will get strange fast

QUESTION TIME

- Solve the following:
  - True or True and True and True
  - True or False and True and True
  - True and not False and not True or False
- Write a function to check whether a number is a multiple of 2 or 3. Print the number only if it is a multiple of both.

Welcome back!

Lets make things slightly more complicated

- We want to write a function that would check whether a given number is greater than 10 and another inputted number y

```
def whoa(x, y):
        if not x > 10 and y:
                print("Yah!")

whoa(3 , 5)
```

Do you see something wrong with this?

Logic Error

- We want a function that tests if the first argument is larger than both the number 10 and also the second argument

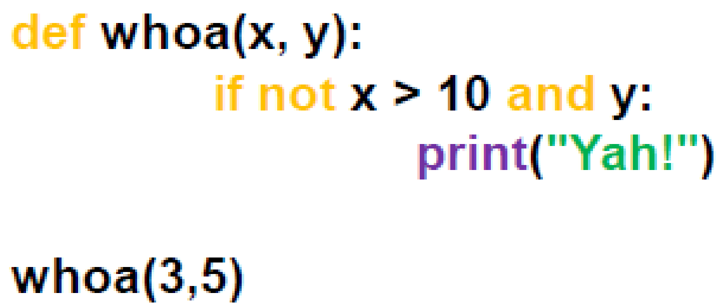
```
def whoa(x, y):
        if not x > 10 and y:
                print("Yah!")
```

should be

```
def whoa(x, y):
        if not x > 10 and x > y:
                print("Yah!")
```

but this raises the question of why it worked at all

Because Python that's why!



```
def whoa(x, y):
        if not x > 10 and y:
                print("Yah!")

whoa(3,5)
```

- this should have evaluated the following conditional:

not x > 10 and y

not 3 > 10 and 5

not False and 5

True and 5

Time for the brain hurt

- Try:
  - print(True + True)

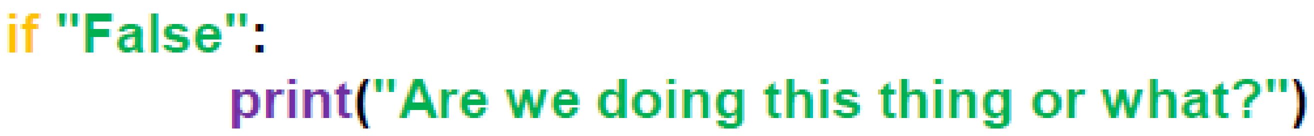
Don't worry I felt the same way.

Booleans as Integers

- The binary nature of boolean values has led to them being treated as 1s and 0s in a lot of languages, and python is no exception.
- So 1 + True is the same as 1 + 1.
- 5/False will give a divide by zero error.
- Consequently you may get a lot of strange behavior if you aren't careful with booleans.

OTHER DATA-TYPES AS BOOLEANS

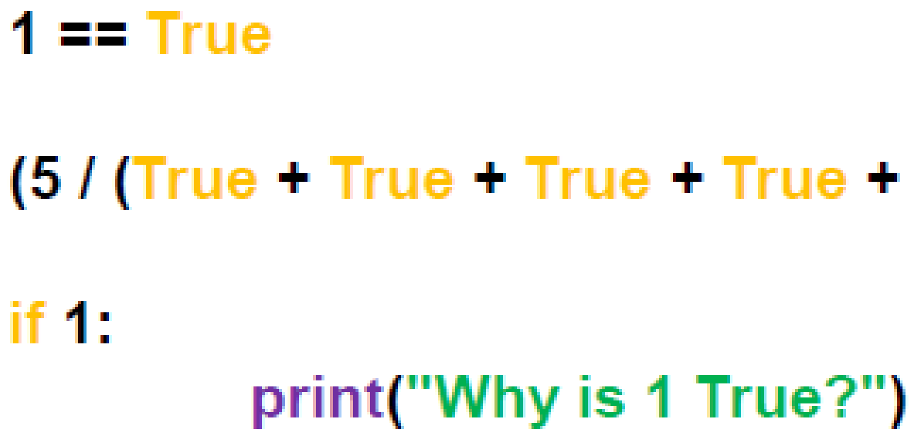
- Any number (**int** or **float**) is treated as **True** unless it has a value of 0 in which case it is treated as **False**
- Similarly any string is treated as **True** unless it is an empty string (i.e. **""**) in which case it is **False**
- so what does this do?



```
if "False":
        print("Are we doing this thing or what?")
```

Some things that work..

- …although it doesn’t seem like they should.



```
1 == True

(5 / (True + True + True + True + True)) == 1

if 1:
        print("Why is 1 True?")
```