#6: Booleans and If Statements

SAMS SENIOR NON-CS TRACK

Last Time

Use **functions** to hold and execute processes

Ex 3-2 Feedback

The functions assignment seems to have been somewhat difficult, especially #3 (parameters) and #4 (returning).

Let's go over how to solve those two problems.

Defining a Function: Example

In general, when we create a function, we want to identify an appropriate **identifier, input, output,** and **process** for that function. These values will directly translate to the function's **name, parameters, return value,** and **body**.

Say we want to define a function that converts money into a number of quarters. Our function components are:

Name: `convertToQuarters`

Parameter: `money`

Body: `numQuarters = money / 0.25`

Result: `return numQuarters`

```
def convertToQuarters(money):
    numQuarters = money / 0.25
    return numQuarters
```

Today's Learning Goals

Understand how **scope** changes where we can access variables

Use **Booleans** to compute whether an expression is True or False

Use **if statements** to make choices about program control flow

Scope

Functions have a different scope

When we define variables inside a function, they **only exist inside the function**. We can't call them in the main code body.

Example:

```
def convertToQuarters(money):
    numQuarters = money / 0.25
    return numQuarters


print(numQuarters * 4) # will crash
```

Scope Organizes Names

This happens because Python considers function bodies to be in a different **scope** that the top-level code. We can only access variables in the scope in which they are defined.

One way to think about this is that a variable's name is its **first** name, but it's function name (or the top level) is its **last** name. You might have the same first name as another person at your school, but you probably have a different last name, and that helps to distinguish between the two of you.

In the example to the right, note that when we print x at the end, it doesn't change to 9 or 11. This is because x top-level is separate from x foo.

```
def foo(x): # This is x foo
    x = x + 2
    return x


x = 5 # This is x top-level


print(f(9))
print(x)
```

You Do: Code Tracing

What will the code to the right print out when we run it?

Try predicting the answer by writing out the steps on paper.

```
def a(x):
    y = 5
    return x + y
def b(x, y):
    return x - y
x = 10
print(a(x) + b(9, 4))
print(x)
print(y)
```

Functions Can Call Functions

We're not restricted to calling functions only at the top-level- we can also **call functions inside of other functions!**

```
def a(x):
    return x * 2
def b(y):
    return a(y) - 1
print(b(10))
```

This lets us write special functions that we'll call **helper functions**. We'll use these when we need to solve large or complicated problems, to break up the work into multiple parts.

The Stack Tracks Function Calls

When a program is calling multiple functions, how do we keep track of which function we're currently in and where the return values should be sent?

Python keeps track of something called a **stack**, which is basically a list of all of the places in the code where we need to eventually return a value. When we reach a return statement, Python removes the current value of the stack and goes back to the previous one.

In the following example, when we've reached line 2, the stack would look like this:

```
1: def a(x):
2:     return x * 2
3: def b(y):
4:     return a(y+1) - 1
5: print(b(10))
```

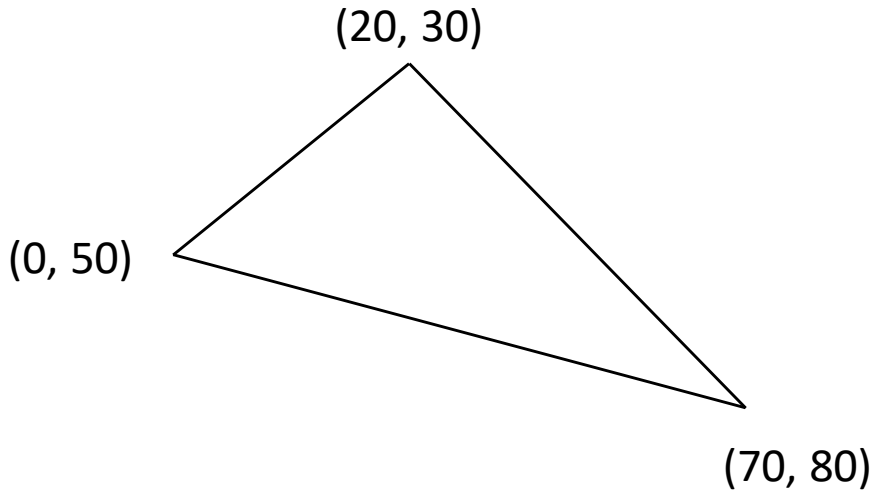
Line 5 – called b() on 10
Line 4 – called a() on 11
Line 2 – return 11 * 2 to previous item in stack

Exercise 1: trianglePerimeter

Exercise 1: write the function trianglePerimeter(x1, y1, x2, y2, x3, y3) which takes three coordinates – (x1, y1), (x2, y2), and (x3, y3) – and calculates the perimeter of the triangle made by connecting those three points.

To make solving this problem easier, you should also write the function distance(x1, y1, x2, y2) which takes two coordinates – (x1, y1) and (x2, y2) – and calculates the distance between them. You should then call distance() from trianglePerimeter().

Finally, print out the result of calling trianglePerimeter on the points (0, 50), (20, 30), and (70, 80) to find the perimeter of that triangle!



Booleans

True-ness and False-ness

So far, we've learned about a few different data types in Python: integers, floats, and strings.

Now we'll learn about another data type that may prove useful: **Booleans**. A Boolean can be one of two values, True or False. These can be typed into a Python expression directly, as you'd type in a variable or a number.

```
print("Yes", True)
print("No", False)
```

Comparisons Create Booleans

We normally create Boolean values by **comparing expressions** in Python. A comparison between two values will always evaluate to True or False based on whether the comparison is correct as stated. Here are some standard examples:

```
print("Less than", 4 < 5) # can also use > for greater than
print("Greater than or equal to", 13 >= (7.2 + 10.3)) # can also use <=
```

Note that the comparison always takes the format <exp1> <operator> <exp2>.

We can also check whether two expressions are exactly equal, or are not equal:

```
print("Equal", (20 - 5) == 19) # note we use two equal signs, not one
print("Not equal", 2 != 0) # note that the ! negates the equality check
```

Comparing Strings

We already know how to compare numbers in real life. Comparing strings is a bit different, but we can do that too!

```
print("Equality", "Hello" != "Goodbye")
```

When we want to see whether one string is less than another, we compare them character-by-character. Each character is associated with an **ASCII value**, and we'll compare those integer values directly. You can get a character's ASCII value by calling `ord(char)`.

```
print("Comparison", "goodbye" < "hello") # "g" comes before "h",
# so "goodbye" comes first
```

ASCII Table

You don't need to memorize ASCII values- you can always look them up in a table.

Note that the digits 0-9, the letters A-Z, and the letters a-z are all in order. This means we can easily compare strings that only contain characters in one of the three groups.

For now, we'll mainly just check equality for strings, not ordering.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Exercise 2: power compare

Exercise 2: at the top level, set up two variables, x and y, that start off holding the values 3 and 5. Then write a line of code that prints out True if x^y is greater than y^x, or False if not.

Note - your code should still work if the numbers inside x and y are changed.

Combining Booleans

We aren't limited to only evaluating a single Boolean expression! We can **combine** Boolean values using **logical operations.** We'll learn about three- **and, or,** and **not**.

Combining Boolean values will let us check complex requirements while running code.

And Operation

The **and** operation takes two Boolean values and evaluates to True if **both** values are True. In other words, it evaluates to False if **either** value is False.

We use **and** when we want to require that both conditions be met at the same time.

and	val1 True	val1 False
val2 True	True	False
val2 False	False	False

Example:

```
(x >= 0) and (x < 10)
```

Or Operation

The **or** operation takes two Boolean values and evaluates to True if **either** value is True. In other words, it only evaluates to False if **both** values are False.

We use **or** when there are multiple valid conditions to choose from

or	val1 True	val1 False
val2 True	True	True
val2 False	True	False

Example:

```
(day == "Saturday") or (day == "Sunday")
```

Not Operation

Finally, the **not** operation takes a single Boolean value and switches it to the opposite value (negates it). not True becomes False, and not False becomes True.

We use **not** to switch the result of a Boolean expression. For example, not (x < 5) is the same as x >= 5.

not	val1 True	val1 False
result	False	True

Example:

```
not (x == 0)
```

Boolean Order of Operations

Like with math operations, Boolean operations will evaluate in a specific order. **not** comes first, then **and**, then **or**. However, it can be a pain to keep track of this ordering while coding.

To make code easier to read, always use parentheses to designate which operations you want to happen first! This is safer than trying to remember how the operations will be ordered.

```
x = 10
print((x > 5) or ((x**2 > 50) and (x == 20))) # True
print(((x > 5) or (x**2 > 50)) and (x == 20)) # False
```

Exercise 3: cloneChecker

Exercise 3: write a function, cloneChecker(name, age), that takes a string (a person's name) and a number (their age). This function returns True if the given name is the same as yours and the age is within one year of yours, or False otherwise.

Then call the function and print out its output twice- first on an input that makes it return True, then on an output that makes it return False.

For example, Prof. Kelly is 30 years old, so for her function, "Kelly" and the age 29, 30, or 31 would result in the code returning True. On the other hand, "Kelly" and the number 18 or "Chloe" and the number 30 would result in the code returning False.

Conditionals

Control Flow

The next few topics we cover will revolve around the idea of **control flow**, or the order in which programming commands are run.

So far, all the code we've written is run **sequentially**. Each line is read and evaluated in order. Functions changed this slightly, but we can still imagine inserting each function's code into the place where the function is called to get step-by-step code.

This next unit will help us write code that is **only executed in certain circumstances**. This lets our code really react to the input that we provide it!

Conditionals

Sometimes we need to change what a program does based on the given input. We can do this using **conditional statements**. These statements choose what the program will do next based on whether or not a boolean expression is True.

```
if <boolean_expression>:
    <body_if_true>
```

Note that, as with functions, conditionals use **indentation** to specify which lines belong to the conditional, and which lines don't. A conditional must have at least one line in the body, but can have more than that as well.

Conditional Example

In the following example, the code will only print "I see you!" if the boolean variable `visible` is set to `True`. However, it will always print "start" and "finish".

```
print("start")
if visible == True:
    print("I see you!")
print("finish")
```

Exercise 4: media test

Exercise 4: at the top level, write a few lines of code that asks the user what their favorite [song/movie/book/tv show] is (just pick one, though!).

If the user's favorite is the same as yours, print out a special message for them. Then, whether or not they had the same favorite, print out a general message about that type of media.

Feel free to get creative with your messages! And if you finish with time to spare, try creating a conversation by adding more inputs and more responses.

For example, Prof. Kelly's current favorite book is Skyward. So if the user inputted a different book (like "The Dark Tower"), her program might print:

```
"I like reading paper books."
```

But if the user inputted "Skyward", her program would print:

```
"I love that book too! Brandon Sanderson is fantastic."
"I like reading paper books."
```

Today's Learning Goals

Understand how **scope** changes where we can access variables

Use **Booleans** to compute whether an expression is True or False

Use **if statements** to make choices about program control flow