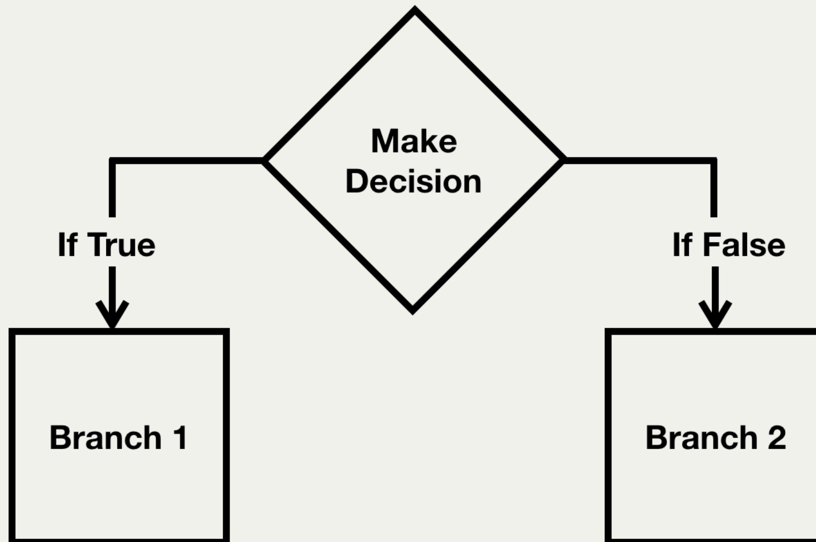
Lecture 3: More on Branching and Iteration, Lists

```
>>> for i in range(10):
...     print(10 - i)
...
10
9
8
7
6
5
4
3
2
1
>>>
```

CS5001 / CS5003:
Intensive Foundations
of Computer Science

```
>>> fib = [0, 1, 1, 2, 3, 5, 8, 13, 21]
>>> for idx, value in enumerate(fib):
...     print("{} : {}".format(idx, value))
...
0 : 0
1 : 1
2 : 1
3 : 2
4 : 3
5 : 5
6 : 8
7 : 13
8 : 21
>>> print(fib[3])
2
>>> print(fib[5:])
[5, 8, 13, 21]
>>> print(fib[-1])
21
>>>
```



PDF of this presentation

Lecture 3: Review: return values

Let's start tonight with a little review! I understand that some students are still unclear about what a *return value* is in a function.

You can think of a return value as the result of a function. Example:

```
>>> import math
>>> s = math.sqrt(25)
>>> print(s)
5.0
>>>
```

The *return value* of the sqrt function is the square root of the parameter. It *returns* the value to the place where it was called.

We can define a function to return whatever we want, and if we call a function and get back a return value, we usually store the return value in a variable (as above). Sometimes, we use the return value immediately, too.

Lecture 3: Review: return values

Here's a function that returns a string:

```
>>> def compliment():
...   num = random.randint(0,2)
...   if num == 0:
...     return "You look beautiful today!"
...   elif num == 1:
...     return "You are so smart!"
...   else:
...     return "I've never met anyone as wonderful as you!"
...
>>> a = compliment()
>>> b = compliment()
>>> c = compliment()
>>> print(a)
You are so smart!
>>> print(b)
You look beautiful today!
>>> print(c)
You are so smart!
>>>
```

With a neighbor, explain what the return value is for this function. Be specific -- how does the function produce the return value?

We will see many functions that return different types of things, and functions returning values is one of the most important aspects of programming.

Lecture 3: if __name__ == "__main__"

What is this `if __name__ == "__main__"` thing all about?

When you run a Python program, it starts reading the file from the first line. If the line starts a function definition, then Python builds the function in memory. If the line is statement, then Python executes that statement. Example:

```
1 def funcA():
2     return "aA"
3
4 print(funcA())
```

On line 1, Python finds a function definition, so it builds up **`funcA`** in memory. Nothing is executed yet.

On line 4, Python finds a statement that calls the print function. It immediately runs it, and prints out "**`aA`**".

Lecture 3: if __name__ == "__main__"

When you run a Python program, Python creates a special variable, called **`__name__`** (also called *dunder name* because of the "double underscore" at the beginning -- these are special Python variables!)

All Python programs that are being run have a default **`__name__`** of "**`__main__`**".

We haven't done this yet, but when a program is *imported into another program*, it has a **`__name__`** equal to its filename (without the .py). So, when we **`import math`** there is a file called **`math.py`** and the name when loaded is **`math`**.

When a program is imported, Python also tries to run it, line by line, just like the initial program that is doing the importing...this can have some consequences!

Therefore, we need to use the **`__name__`** variable to determine what gets run in our program.

Lecture 3: if __name__ == "__main__"

Two example programs:

programA.py:

```
1 def whoami():
2     return "I'm programA!"
3
4 print("I am a rude programA.")
```

programB.py

```
1  def whoami():
2      return("I'm programB!")
3
4  if __name__ == "__main__":
5      import programA
6      a = programA.whoami()
7      b = whoami()
8
9      print("a == {}".format(a))
10     print("b == {}".format(b))
```

What prints out when we run programA.py? (talk to your neighbor!)

What prints out when we run programB.py? (talk to your neighbor!)

Is there a problem? Did programB.py run as the programmer might have expected it to run?

Lecture 3: if __name__ == "__main__"

A third program:

programB.py

```python
def whoami():
    return("I'm programB!")

if __name__ == "__main__":
    import programA
    a = programA.whoami()
    b = whoami()

    print("a == {}".format(a))
    print("b == {}".format(b))
```

programC.py

```python
import programB

def whoami():
    return("I'm programC!")

if __name__ == "__main__":
    b = programB.whoami()
    c = whoami()

    print("b == {}".format(b))
    print("c == {}".format(c))
```

What prints out when we run programC.py? (talk to your neighbor!)

Do you see how this is actually better because both programs use the **`if __name__ == "__main__"`** conditional statement?

Lecture 3: A new kind of Python string: the f-string!

In Lecture 1, we learned about string formatting, using string.format to make a string include variables, mostly for printing.

It turns out that Python 3.6 introduced an even better way to format strings, called *f-strings*. f-strings, technically called "format string literals" allows you to put what you want to format *directly* into the curly braces. This can make the string much easier to read. Example:

```
if __name__ == "__main__":
    name = input("What is your name? ")
    age = int(input("What is your age? "))
    fav_superhero = input("What is your favorite superhero? ")
    last_book_read = input("What was the last book you read? ")

    print(f"Hello, {name}! You were born in {2019 - age}.")
    print(f"What would '{last_book_read}' be like if the main character was "
          f"replaced with {fav_superhero}?")
```

```
What is your name? Chris
What is your age? 21
What is your favorite superhero? Wonder Woman
What was the last book you read? To Kill a Mockingbird
Hello, Chris! You were born in 1998.
What would 'To Kill a Mockingbird' be like if the main character was replaced with Wonder Woman?
```

Lecture 3: A new kind of Python string: the f-string!

Feel free to use f-strings in your own code.

To use an f-string, you must prefix the string itself with **`f`**

```
>>> a = 5
>>> b = 6
>>> correct = f"The sum of {a} + {b} == {a + b}"
>>> incorrect = "The sum of {a} + {b} == {a + b}"
>>> print(correct)
The sum of 5 + 6 == 11
>>> print(incorrect)
The sum of {a} + {b} == {a + b}
```

Again, don't forget to put the **`f`** before the string!

See here for more details about formatting and f-strings:

https://realpython.com/python-f-strings/

Lecture 3: More on conditionals

Last week, we introduced the *conditional* statement, **`if`**. You practiced using conditionals in lab and on your homework assignment.

Today, we will expand the use of the **`if`** statement to include:

- the use of the *boolean* operators: **`not`**, **`and`**, and **`or`**.
- *chaining* conditional statements
- the *membership* operator, **`in`**

Remember, all conditionals used in an **`if`** statement must be either **`True`** or **`False`**.

Lecture 3: More on conditionals: not, and, or

The **`not`** operator takes a boolean expression and makes it the opposite. If an expression is **`True`** it becomes **`False`**, and if an expression is **`False`**, it becomes **`True`**. Examples:

```
1  >>> 4 < 5
2  True
3  >>> not 4 < 5
4  False
5  >>> 5 < 4
6  False
7  >>> not 5 < 4
8  True
9  >>> 3 == 3
10 True
11 >>> not 3 == 3
12 False
```

All of these expressions with **`not`** have equivalents that you could use instead of **`not`**. For example, instead of saying "**`not 4 < 5`**" you could say, "**`4 >= 5`**" So, we rarely use **`not`** for simple cases like this, and use it more often for longer expressions that we want to invert.

Here is a *truth table* that shows how **`not`** works:

a	not a
True	False
False	True

Lecture 3: More on conditionals: not, and, or

The **`and`** operator takes to expressions and performs a logical **`and`** with them. In other words, the entire expression is **`True`** only if the first expression *and* the second espression are *both* **`True`**. Here is a truth table that shows this:

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

Examples:

```
 1 >>> (4 < 5) and (6 < 7)
 2 True
 3 >>> (4 < 5) and (7 < 6)
 4 False
 5 >>> (5 < 4) and (6 < 7)
 6 False
 7 >>> (5 < 4) and (7 < 6)
 8 False
 9 >>> (4 < 5) and (5 < 6) and (6 < 7)
10 True
11 >>> (4 < 5) and (5 < 6) and (6 < 7) and (9 < 8)
12 False
13 >>>
```

You can have as many expressions as you want, with as many **`and`**s as you want.

You generally want to put parentheses around expressions to make it clear.

Lecture 3: More on conditionals: not, and, or

The **`or`** operator takes to expressions and performs a logical **`or`** with them. In other words, the entire expression is **`True`** if *any* of the expressions are **`True`**. Here is a truth table that shows this:

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

Examples:

```
1 >>> (4 < 5) or (6 < 7)
2 True
3 >>> (4 < 5) or (7 < 6)
4 True
5 >>> (5 < 4) or (6 < 7)
6 True
7 >>> (5 < 4) or (7 < 6)
8 False
9 >>> (5 < 4) or (7 < 6) or (9 < 8) or (10 < 11)
10 True
```

You can have as many expressions as you want, with as many **`or`**s as you want.

You generally want to put parentheses around expressions to make it clear.

Lecture 3: More on conditionals: not, and, or

Example program using **`and`**, **`or`**, and **`not`**:

```
def str2bool(v):
    # we'll learn the 'in' operator later tonight!
    return v.lower() in ("yes", "true", "t", "1")


if __name__ == "__main__":
    print("In boolean logic, 'DeMorgan's Theorems' say the following:")
    print("Theorem 1:")
    print("\tA and B == not (not A or not B)\n")
    print("Theorem 2:")
    print("\tA or B == not (not A and not B)\n")

    print("Let's test to see if this is correct.")
    a = str2bool(input("Do you want A to be True or False? "))
    b = str2bool(input("Do you want B to be True or False? "))
    print(f"A and B == {a and b}")
    print(f"not (not A or not B) == {not (not a or not b)}")
    print()

    print(f"A or B = {a or b}")
    print(f"not (not A and not B) == {not (not a and not b)}")
    print()

    if (a and b) == (not (not a or not b)) and (a or b) == (not (not a and
                                                                 not b)):
        print("See -- it works!")
```

Lecture 3: More on conditionals: multiple comparisons

If you want to make a magnitude comparison between two amounts, you can do so as follows:

```
>>> a = 20
>>> if 10 < a < 30:
...     print("a is between 10 and 30")
...
a is between 10 and 30
>>> b = 10
>>> if 10 < b < 30:
...     print("b is between 10 and 30")
... else:
...     print("b is not between 10 and 30")
...
b is not between 10 and 30
>>>
```

Lecture 3: Iteration

In the last lecture, we ended class with an example game where a player tries to guess a random number that the computer chose. We allowed the player to have three guesses:

```
print("I have chosen a number between 0 and {}, inclusive.".format(maximum))

    guess = int(input("Try to guess my number. You have 3 tries left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    guess = int(input("Try to guess my number. You have 2 tries left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    guess = int(input("Try to guess my number. You have 1 try left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    print("The number I chose was {}.".format(computer_choice))
```

I posed the question: what if we wanted to give the player 5, 10, or 100 guesses? What would we have to do?

Lecture 3: Iteration

```
    print("I have chosen a number between 0 and {}, inclusive.".format(maximum))

    guess = int(input("Try to guess my number. You have 3 tries left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    guess = int(input("Try to guess my number. You have 2 tries left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()
    ...
```

I posed the question: what if we wanted to give the player 5, 10, or 100 guesses? What would we have to do?

With what we know so far, we would have to duplicate our code 5, 10, or 100 times! If we wanted to allow an *infinite* number of guesses, we couldn't even program that, yet!

So, we need to introduce a new concept: *iteration* (also called, *looping*)

Iteration is *the repeated execution of a set of statements*. Once we can do that, the problem posed above becomes doable!

Lecture 3: Iteration

In Python, there are many ways to iterate. One way is through the use of a **`while`** statement:

```
>>> while True:
...     print("This goes on forever!")
...
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
This goes on forever!
...(eventually I have to type ctrl-c to stop it)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
>>>
```

The **`while`** statement evaluates the expression, and if it is **`True`** (which it always is in the example to the left), then it starts executing the block.

When the block is completed, the **`while`** statement is re-evaluated, and if the expression is still **`True`** then the block is executed again.

The *iteration* continues until the expression in the **`while`** statement becomes **`False`**.

Lecture 3: Iteration

Here is another example of the **`while`** statement. In this case, we are counting down from 10 to 1.

```
if __name__ == "__main__":
    i = 10
    while i > 0:
        print(f"Counting down...{i}")
        i = i - 1
    print("Blastoff!")
```

Output:

```
Counting down...10
Counting down...9
Counting down...8
Counting down...7
Counting down...6
Counting down...5
Counting down...4
Counting down...3
Counting down...2
Counting down...1
Blastoff!
```

On the next slide, we will discuss what the `i = i - 1` means specifically, but it simply changes the value of `i` to one less each iteration of the while loop. On the first iteration, `i = 10`, on the second iteration, `i = 9`, etc., all the way to `0`.

Each time, the while loop checks to see that `i > 0`, and when that becomes **`False`**, it does not execute the block.

Lecture 3: Changing the value of a variable

```
if __name__ == "__main__":
    i = 10
    while i > 0:
        print(f"Counting down...{i}")
        i = i - 1
    print("Blastoff!")
```

The code above has the following line:

```
i = i - 1
```

This does not seem to make sense mathematically!

But, remember that in Python, a single equals sign means *assignment*. This can be confusing to new programmers. To be clear, when you say an assignment out loud, it is sometimes better to say "variable *gets* value" instead of "variable *equals* value". For example, for the following, I would say, "v *gets* 42" for the first line of code:

```
>>> v = 42
>>> print(v)
42
```

Lecture 3: Changing the value of a variable

```
if __name__ == "__main__":
    i = 10
    while i > 0:
        print(f"Counting down...{i}")
        i = i - 1
    print("Blastoff!")
```

So, how do we explain this?

```
i = i - 1
```

This means, "`i` *gets* the *current value* of `i` minus `1`."

So, if the current value of `i` is `10`, then after this statement, the value of `i` is now `9`.

As described on the last slide, `i` keeps going down by `1` each iteration of the `while` loop, because the `i = i - 1` statement does just that.

Lecture 3: Changing the value of a variable

```
if __name__ == "__main__":
    i = 10
    while i > 0:
        print(f"Counting down...{i}")
        i -= 1
    print("Blastoff!")
```

There is a slight change in the program above, but it runs exactly the same.

There is a shorthand for the following form:

```
var = var - x
```

The following is equivalent to the above statement:

```
var -= x
```

You can do the same for most arithmetic reassignments. Here is a table:

Operator	Full statement	Shorthand	If v was originally 25, now v is
+	v = v + 5	v += 5	30
-	v = v - 5	v -= 5	20
*	v = v * 5	v *= 5	125
/	v = v / 5	v /= 5	5.0

Lecture 3: Player-chosen number of guesses

Now that we know about the **`while`** loop, we can modify our game program to make it both shorter, and more interesting!

This is how the program will run after our changes:

```
I have chosen a number between 0 and 18, inclusive.
How many attempts would you like? 5
Try to guess my number. You have 5 tries left. 9
Too low!
 Guess again.
Try to guess my number. You have 4 tries left. 14
Too high!
 Guess again.
Try to guess my number. You have 3 tries left. 12
Too high!
 Guess again.
Try to guess my number. You have 2 tries left. 10
Too low!
 Guess again.
Try to guess my number. You have 1 try left. Good luck! 11
You guessed my number!
Goodbye!
```

We are going to let the player choose how many choices they would like, and we are going to use a **`while`** loop to make that happen.

Lecture 3: Player-chosen number of guesses

Let's ask the player for the number of tries, and then have a **`while`** loop that counts down from that number of tries:

```
num_tries = int(input("How many attempts would you like? "))

while num_tries > 0:
    guess = int(input(f"Try to guess my number. You have {num_tries} tries "
                      "left. "))
    if evaluate_guess(computer_choice, guess) == 0:
        quit()

    num_tries -= 1
```

Now, even if the user wants 1 million tries, they can get them!

The way we have this written, there will be a typo when the user gets to the last guess: "You have 1 tries left." What can we do about this?

Lecture 3: Special cases

We can have a *special case* for the value when **`num_tries`** is **`1`**:

```
while num_tries > 0:
        if num_tries == 1:
            guess = int(input(f"Try to guess my number. You have 1 try "
                              f"left. Good luck! "))
        else:
            guess = int(input(f"Try to guess my number. You have {num_tries} "
                              f"tries left. "))
        if evaluate_guess(computer_choice, guess) == 0:
            quit()

        num_tries -= 1
```

Often in programming, you will have to special case a particular code path. There are ways to minimize the duplicated code, but there will always be more logic that you have to put into the program.

Lecture 3: Special cases

Here is another common special case: if you have a list of

```
while num_tries > 0:
        if num_tries == 1:
            guess = int(input(f"Try to guess my number. You have 1 try "
                              f"left. Good luck! "))
        else:
            guess = int(input(f"Try to guess my number. You have {num_tries} "
                              f"tries left. "))
        if evaluate_guess(computer_choice, guess) == 0:
            quit()

        num_tries -= 1
```

Often in programming, you will have to special case a particular code path. There are ways to minimize the duplicated code, but there will always be more logic that you have to put into the program.

Lecture 3: The for in range() statement

The **`while`** loop provides a basic form of iteration, and there is another type of iteration called a **`for`** loop. The most common **`for`** loop uses the **`range`** *class* to get values to iterate through. Ranges are, simply, lists of numbers in order. Here is an example using a **`for`** loop and a **`range`**:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

`range(5)` will produce the numbers 0, 1, 2, 3, and 4, in order. Why do we start with 0? Computer scientists traditionally start numbering from 0, because it makes *indexing* into a list a bit easier then starting with 1.

You will quickly get used to starting numbering at 0 as you learn programming!

Why does it stop at 4? This is a bit historical, but for a range of 5, we want just 5 values, and if we start at 0, we have: 0, 1, 2, 3, 4, which is 5 values.

Lecture 3: The for in range() statement

The **`range`** class defaults to starting with 0, but we can change that with a second parameter, or a third.

In the first example, below, **`range(2, 5)`** means to *start* at 2 and end at 4 (so there are a total of 3 values).

In the second example, the last parameter is the *step* -- it is added to the previous value to get the next value:

```
>>> for i in range(2, 5):
...   print(i)
...
2
3
4

>>> for i in range(2, 10, 3):
...   print(i)
...
2
5
8
>>>
```

In the second example, you can also see that it stops at 8. This is because the next possible value, 11, is greater than 9, which is where the range would stop without a step (or with a step of 1).

Let's practice some ranges in the REPL!

Lecture 3: The for in range() statement

The **`range`** class also allows us to have a range that goes *backwards*:

```
>>> for i in range(5,0,-1):
...   print(i)
...
5
4
3
2
1
```

Now, the step is -1, making the range go down in values. Notice that the range stops at 1, which is one before the stop value (just like when going upwards).

If you wanted to have a range that was exactly the reverse of range(5), you would do the following:

```
>>> for i in range(4,-1,-1):
...   print(i)
...
4
3
2
1
0
```

You can also have a negative step that is not -1:

```
>>> for i in range(10,5,-2):
...   print(i)
...
10
8
6
```

Lecture 3: The for in range() statement

The **`for in range`** statement is useful if you know exactly how many times you are going to go through the loop. We can change our guessing game code to use a **`for`** loop instead of a **`while`** loop, because we know the number of times we will loop:

```
    for tries_left in range(num_tries, 0, -1):
        if tries_left == 1:
            last_guess = True
            guess = int(input(f"Try to guess my number. You have 1 try "
                              f"left. Good luck! "))
        else:
            guess = int(input(f"Try to guess my number. You have {tries_left} "
                              f"tries left. "))
        if evaluate_guess(computer_choice, guess, last_guess) == 0:
            quit()
```

Notice that we don't need to manually update **`tries_left`** anywhere in particular (like we would have had to do in a **`while`** loop). **`tries_left`** is updated each iteration via the **`range`** calculation.

Lecture 3: Lists

One of the fundamental *data structures* that Python has is the *list*. A Python list is exactly what it sounds like: a list of values. Example:

```
>>> instructor_names = ['Chris', 'Mark', 'Joyce', 'Yiya']
>>> print(instructor_names)
['Chris', 'Mark', 'Joyce', 'Yiya']
>>> for name in instructor_names:
...   print(f"{name} is an instructor.")
...
Chris is an instructor.
Mark is an instructor.
Joyce is an instructor.
Yiya is an instructor.
```

As in the example above, we can iterate through a list, just like using the **`range`** function.

To create a list, you simply put the values into square brackets, separated by commas:

```
san_cities = ['San Francisco', 'San Jose', 'San Diego', 'San Luis Obispo', 'San Onofre']
```

Lecture 3: Lists

Lists can have different types in them (see below), but it is almost always a good idea to stick to one type of element:

```
>>> my_list = [12, 1.2, 'Northeastern', [1,2,3]]
>>> for element in my_list:
...   print(f"{element} is {type(element)}")
...
12 is <class 'int'>
1.2 is <class 'float'>
Northeastern is <class 'str'>
[1, 2, 3] is <class 'list'>
```

Notice that a list can have another list as an element! This is called a *nested* list. Here is an example:

```
>>> points = [[1,2], [5,8], [-2,7], [-5, -5]]
>>> for point in points:
...   print(point)
...
[1, 2]
[5, 8]
[-2, 7]
[-5, -5]
```

Lecture 3: Lists: accessing elements

If you want to access a particular element in a list, you use *bracket notation*. Elements are numbered, starting from 0 (that is important!). The first element has the value **`my_list[0]`**, the second has the value **`my_list[1]`**, etc.

```
>>> san_cities = ['San Francisco', 'San Jose', 'San Diego', 'San Luis Obispo', 'San Onofre']
>>> san_cities[0]
'San Francisco'
>>> san_cities[1]
'San Jose'
>>> san_cities[4]
'San Onofre'
>>> san_cities[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

If you try to access a list element that is *out of range* (too big), you get a runtime error!

Lecture 3: Lists: accessing elements

You can change a list element by referring to it with bracket notation:

```
>>> san_cities = ['San Francisco', 'San Jose', 'San Diego', 'San Luis Obispo', 'San Onofre']
>>> san_cities[0] = 'San Bruno'
>>> san_cities
['San Bruno', 'San Jose', 'San Diego', 'San Luis Obispo', 'San Onofre']
>>> san_cities[4] = 'San Mateo'
>>> san_cities
['San Bruno', 'San Jose', 'San Diego', 'San Luis Obispo', 'San Mateo']
>>> san_cities[5] = 'San Antonio'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

You cannot set a value that doesn't already exist in the list (as seen on the last attempt, above)

However, you can *append* a value, which puts a new element at the end of the list:

```
>>> san_cities.append('San Antonio')
>>> san_cities
['San Bruno', 'San Jose', 'San Diego', 'San Luis Obispo', 'San Mateo', 'San Antonio']
```

Lecture 3: Lists: accessing elements

If you want to add an element somewhere else in the list, you can do so with the *insert* method:

```
>>> help(san_cities.insert)
Help on built-in function insert:

insert(index, object, /) method of builtins.list instance
    Insert object before index.

>>> san_cities.insert(0,'San Rafael')
>>> san_cities
['San Rafael', 'San Bruno', 'San Jose', 'San Diego', 'San Luis Obispo', 'San Mateo',
'San Antonio']
>>> san_cities.insert(3,'San Leandro')
>>> san_cities
['San Rafael', 'San Bruno', 'San Jose', 'San Leandro', 'San Diego', 'San Luis Obispo',
'San Mateo', 'San Antonio']
```

Lecture 3: Lists: accessing elements

If you want to know how many elements are in a list, you use the built-in function, **`len`**:

```
>>> san_cities
['San Rafael', 'San Bruno', 'San Jose', 'San Leandro', 'San Diego', 'San Luis Obispo',
'San Mateo', 'San Antonio']
>>> len(san_cities)
8
```

Note that **`len`** is not a method you use with dot notation -- it is simply a function you call with the list as the argument.

Question: what is the index of the last element in a list, based on its length?

Lecture 3: Lists: accessing elements

If you want to know how many elements are in a list, you use the built-in function, **`len`**:

```
>>> san_cities
['San Rafael', 'San Bruno', 'San Jose', 'San Leandro', 'San Diego', 'San Luis Obispo',
'San Mateo', 'San Antonio']
>>> len(san_cities)
8
```

Note that **`len`** is not a method you use with dot notation -- it is simply a function you call with the list as the argument.

Question: what is the index of the last element in a list, based on its length?

Answer: **`len(list_name) - 1`**

```
>>> san_cities
['San Rafael', 'San Bruno', 'San Jose', 'San Leandro', 'San Diego', 'San Luis Obispo',
'San Mateo', 'San Antonio']
>>> len(san_cities)
8
>>> san_cities[len(san_cities) - 1]
'San Antonio'
```

Lecture 3: Lists: accessing elements

You can also refer to the last element in a list with as **`list_name[-1]`**

```
>>> san_cities
['San Rafael', 'San Bruno', 'San Jose', 'San Leandro', 'San Diego', 'San Luis Obispo',
'San Mateo', 'San Antonio']
>>> san_cities[-1]
'San Antonio'
```

You can access all the elements in reverse order by their negative-number counterparts:

```
>>> san_cities
['San Rafael', 'San Bruno', 'San Jose', 'San Leandro', 'San Diego', 'San Luis Obispo',
'San Mateo', 'San Antonio']
>>> san_cities[-2]
'San Mateo'
>>> san_cities[-7]
'San Bruno'
>>> san_cities[-8]
'San Rafael'
>>> san_cities[-9]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

Just be careful not to go below **`-len(list_name)`**

Lecture 3: Lists: The zip function

If you have two lists and you want to iterate through them together, you can do so with the `zip` function:

```
>>> cities = ['Anchorage', 'Miami', 'Boston']
>>> populations = [294356, 463347, 685094]
>>> for city, pop in zip(cities, populations):
...   print(f"{city} has a population of {pop}.")
...
Anchorage has a population of 294356.
Miami has a population of 463347.
Boston has a population of 685094.
```

If one list has fewer elements, then the iteration stops when the shorter list runs out:

```
>>> cities.append('New York')
>>> cities
['Anchorage', 'Miami', 'Boston', 'New York']
>>> populations
[294356, 463347, 685094]
>>> for city, pop in zip(cities, populations):
...   print(f"{city} has a population of {pop}.")
...
Anchorage has a population of 294356.
Miami has a population of 463347.
Boston has a population of 685094.
```

Lecture 3: Recap

What we covered tonight:

- Return value review
- `if __name__ == "__main__":`
- f-strings for easier formatting
- More on conditionals
  - `not`, `and`, `or`
  - multiple operators (e.g., `if 10 < a < 20`:)
- Iteration
  - `while` loops
  - `for in` loops
- Changing a variable's value
- Lists
  - Creating lists
  - Accessing elements
  - The `zip` function