

CODE CAMP

Day 003: Lists and Loops

Dashboard

CPU Usage: 0% used – 0.00s of 100s. Resets in 23 hours, 15 minutes [More](#)

File storage: 0% full – 124.0 MB of 512.0 MB quota

Recent Consoles

+ 5 -

 [Bash console 15154103](#)

 [Bash console 15153805](#)

New console:

You can have up to 2 consoles. To get more, [upgrade your account!](#)

Recent Files

You have no recently edited files

[+ Open another file](#)

 B

.001 Log In to Python Anywhere

Visit [PythonAnywhere.com](https://pythonanywhere.com) and log in to your account. You can click on one of the bash consoles you logged in to previously. They are listed under “Recent Consoles.”

If your command line is cluttered from your previous work you can type the **clear** command to clear the console visual history.



.001 Log In to Python Anywhere

If you've done everything correctly, your screen should appear similar to the image above. At your command prompt, type the command **nano** to open your text editor.

```
family = ["Brett", "Kerry", "Joan", "Rick",  
"Colleen", "Connor"]  
  
computers = ("Macintosh", "IBM PC", "Atari 800",  
"Commodore 64", "TRS-80")
```

.002 Creating a List (Array)

A list is a group of items that is both ordered and mutable (changeable). A list may have the same item contained in more than one location.

A Tuple is similar to a list, except, once declared it is immutable (unchangeable).

A list is declared with hard brackets as in **family** on the left.

A tuple is declared with parentheses as on the example **computers** on the left.

```
family = ["Brett", "Kerry", "Joan", "Rick",  
"Colleen", "Connor"]  
  
computers = ("Macintosh", "IBM PC", "Atari 800",  
"Commodore 64", "TRS-80")  
  
print (family)  
print (computers)
```

.002 Creating a List (Array)

You can use the **print** command to output the list or tuple in its entirety. However, you would not ordinarily simply print out an entire list or tuple.


Reload this page Console 15154103

```
15:59 ~ $ nano
16:03 ~ $ python list.py
['Brett', 'Kerry', 'Joan', 'Rick', 'Colleen', 'Connor']
('Macintosh', 'IBM PC', 'Atari 800', 'Commodore 64', 'TRS-80')
16:03 ~ $ nano list.py
16:04 ~ $
```

.002 Creating a List (Array)

You will notice that when you print out the list it is surrounded by hard brackets and when you print the tuple it is surrounded by parentheses.

While the list and tuple are collection of elements, you'll normally only work with a single data point at a time.

```
family = ["Brett", "Kerry", "Joan", "Rick",  
"Colleen", "Connor"]  
  
computers = ("Macintosh", "IBM PC", "Atari 800",  
"Commodore 64", "TRS-80")  
  
print (family[0])  
print (family[1])  
print (computers[0])  
print (computers[3])
```

.003 Accessing List Members

You can access individual members of a list or tuple by it's index (or position) in the list. Like most programming languages, Python is zero-indexed. That means the first member of a list or tuple is member 0. You would address it as:

family[0]

Enter the code on the right. You may substitute the names of your own family members.



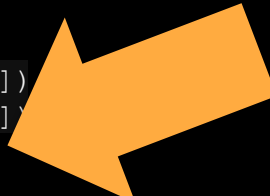
```
16:11 ~ $ python list.py
Brett
Kerry
Macintosh
Commodore 64
16:11 ~ $
```

.003 Accessing List Members

Run the script you just created in your command line. You will note that the members of the list and tuple are accessed by their index:

Brett	Index 0
Kerry	Index 1
Macintosh	Index 0
Commodore 64	Index 3


```
family = ["Brett", "Kerry", "Joan", "Rick",  
          "Colleen", "Connor"]  
  
computers = ("Macintosh", "IBM PC", "Atari 800",  
            "Commodore 64", "TRS-80")  
  
print (family[0])  
print (family[1])  
print (computers[0])  
print (computers[3])  
print (family[10])
```



```
Traceback (most recent call last):  
  File "list.py", line 9, in <module>  
    print (family[10])  
IndexError: list index out of range  
16:15 ~ $
```

.003 Accessing List Members

Note that attempting to access a non-existent index will cause an error.

You can, however, access a range of indexes in a list or tuple using the range operator like this:

```
print (family[0:3])
```

This code will start at index 0 and stop *before* index 3.

In python : is used as the range operator.

```
family = ["Brett", "Kerry", "Joan", "Rick",  
"Colleen", "Connor"]  
computers = ("Macintosh", "IBM PC", "Atari 800",  
"Commodore 64", "TRS-80")  
  
print (family[0])  
print (family[1])  
print (computers[0])  
print (computers[3])  
print (family[0:2])
```

.003 Accessing List Members

Notice in this example, the values returned are index 0 and index 1. The range operator stops returning values *before* index 2.


In a list, you can replace members of the list with the assignment operator. If I wanted to, with apologies, replace my brother Brett with myself the could would be:

family[0] = "Mark"




Bash console 15154103

```
16:18 ~ $ python list.py  
Brett  
Kerry  
Macintosh  
Commodore 64  
['Brett', 'Kerry']  
16:18 ~ $
```

A large orange arrow pointing from the right towards the output of the Python script, specifically highlighting the list output.

```
family = ["Brett", "Kerry", "Joan", "Rick",  
"Colleen", "Connor"]  
computers = ("Macintosh", "PC", "Atari 800",  
"Commodore 64", "TR")  
  
family[0] = "Mark"  
  
print (family[0])  
print (family[1])  
print (computers[0])  
print (computers[3])  
print (family[0:2])
```



.003 Accessing List Members


When this code is executed, the original value in **family** index 0 is replaced with "Mark". This is reflected in the output when the program is executed.



Bash console 15154103

```
16:23 ~ $ python list.py  
Mark  
Kerry  
Macintosh  
Commodore 64  
['Mark', 'Kerry']  
16:23 ~ $
```

```
family = ["Brett", "Kerry", "Joan", "Rick",  
"Colleen", "Connor"]  
computers = ("Macintosh", "IBM PC", "Atari 800",  
"Commodore 64", "TRS-80")  
  
family[0] = "Mark"  
computers[0] = "Amiga"  
  
print (family[0])  
print (family[1])  
print (computers[0])  
print (computers[3])  
print (family[0:2])
```



.003 Accessing List Members

Attempting to change a value in a tuple will always result in an error. In this code example, **computers**, is a tuple and when we attempt to reassign the value of index 0, an error is generated.

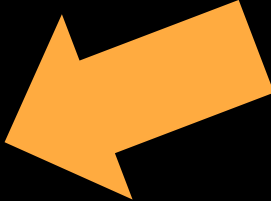
Read errors carefully, as they often provide valuable information that allows you to easily debug your program.



Bash console 15154103

```
16:26 ~ $ python list.py  
Traceback (most recent call last):  
  File "list.py", line 6, in <module>  
    computers[0] = "Amiga"  
TypeError: 'tuple' object does not support item assignment  
16:26 ~ $
```

```
family = ["Brett", "Kerry", "Joan", "Rick",  
"Colleen", "Connor"]  
  
computers = ("Macintosh", "IBM PC", "Atari 800",  
"Commodore 64", "TRS-80")  
  
family[0] = "Mark"  
  
family.append("Brett")  
family.remove("Mark")  
family.insert(0, "Carol")  
print(family)
```



.004 Manipulating Lists

There are a number of commands that can be used to manipulate the values in lists. Enter the code on the left to try these commands:

append()	Adds to end of list
remove()	Removes list member
insert(x,x)	Inserts at index

For **insert** the first argument is the index and the second, the data to insert.

In our example, we first add **Brett** to the end of the list. We then remove **Mark** from index 0. Finally we add **Carol** at index 0. Examine the Bash console output. You'll notice the order reflects the commands issues.



Bash console 15154103

```
16:33 ~ $ python list.py  
['Carol', 'Kerry', 'Joan', 'Rick', 'Colleen', 'Connor', 'Brett']  
16:33 ~ $
```

Table of Contents

- 5. Data Structures
 - 5.1. More on Lists
 - 5.1.1. Using Lists as Stacks
 - 5.1.2. Using Lists as Queues
 - 5.1.3. List Comprehensions
 - 5.1.4. Nested List Comprehensions
 - 5.2. The `del` statement
 - 5.3. Tuples and Sequences
 - 5.4. Sets
 - 5.5. Dictionaries
 - 5.6. Looping Techniques
 - 5.7. More on Conditions
 - 5.8. Comparing Sequences and Other Types

Previous topic

4. More Control Flow Tools

5. Data Structures ¶

This chapter describes some things you've I things as well.

5.1. More on Lists

The list data type has some more methods. ¶

`list.append(x)`

Add an item to the end of the list. Equiva

`list.extend(iterable)`

Extend the list by appending all the `iterable`.

`list.insert(i, x)`

Insert an item at a given position. The fi
insert, so `a.insert(0, x)` inserts at the
to `a.append(x)`.

`list.remove(x)`

Remove the first item from the list whose

.004 Manipulating Lists

There are actually a number of commands that can be used with lists. It is important to be able to read the official Python documentation to fully understand the capabilities of the language.

Visit the Python documentation now at <https://docs.python.org/3/tutorial/datastructures.html> to explore more on lists.

```
family = ["Brett", "Kerry", "Joan", "Rick",  
"Colleen", "Connor"]  
  
computers = ("Macintosh", "IBM PC", "Atari 800",  
"Commodore 64", "TRS-80")  
  
for x in family:  
    print (x)  
  
for computer in computers:  
    print (computer)
```

.005 Looping Through a List

The **for** commands allows you to loop through all the members of a list or tuple. As it loops, with each loop iteration the current member is assigned to the variable indicated.

In our example as we loop the value of **x** is first **Brett**, then it's **Kerry**, then **Joan**, and so on.



Bash console 15154103

```
16:45 ~ $ python list.py  
Brett  
Kerry  
Joan  
Rick  
Colleen  
Connor  
Macintosh  
IBM PC  
Atari 800  
Commodore 64  
TRS-80
```

```
towns = ("Westport", "Trumbull", "Hartford",  
         "Milford", "Fairfield", "Madison", "New Haven",  
         "Chicago", "Houston", "Las Vegas")  
  
for town in towns:  
    print town
```



Bash console 15154103

```
16:53 ~ $ python listloop.py  
Westport  
Trumbull  
Hartford  
Milford  
Fairfield  
Madison  
New Haven  
Chicago  
Houston  
Las Vegas  
16:53 ~ $
```

.005 Looping Through a List

Try entering the example on the left in to Nano.

As the loop execute, each member of the tuple is assigned to the variable **town** and printed. (Remember the tuple itself is called **towns**.)

Ensure your results are the same as displayed on the left.

.006 While Loops

While loops allow you to run a block of code **while** some condition is true. Loops are a very common coding structure, and, if you think about it, many things that occur during a programs' execution are a loop.

For examples think of a program that simulates dealing a deck of cards or a game that waits for the user to do something, reacts, and waits again.

We'll begin with simple loops that count.

```
x = 0
while x < 100:
    print x
    x = x + 5
```



Bash console 15154103

```
16:59 ~ $ python loops.py
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
16:59 ~ $
```

.005 While Loops

Start a new buffer in nano and type in the code on the left. Save the file as **loop.py** and run in on your console.

When the loop beings the value of **x** is 0. The text is run and **x < 100** resolves as true so the loop continues because **0 < 100**. The value of **x** is printed and 5 is added to **x**. **x** is now 5 and the test is performed again.

This continues until **x** contains the value 95. 5 is added and the value of **x** is now 100. When the test is conducted, **x < 100**, evaluates as false because **100 < 100** is not true. At this opnt the loop exits and the program terminates.

```
teams = []
x = ""
while x != "xxx":
    x = ""
    print("Enter a team or xxx to quit")
    x = raw input()
    print (x + " added.")
    if x != "xxx":
        teams.append(x)
print("Here are the teams you entered:")
for team in teams:
    print team
```

.005 While Loops

Start a new nano buffer and key in the example on the left.

In this example we're using a **while** loop differently. The **while** loop is being used to take multiple inputs.

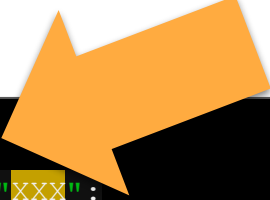
Note that the symbol **!=** means not equal to.



```
17:12 ~ $ python loops.py
Enter a team or XXX to quit
Yankees
Yankees added.
Enter a team or XXX to quit
Mets
Mets added.
Enter a team or XXX to quit
Giants
Giants added.
Enter a team or XXX to quit
Jets
Jets added.
Enter a team or XXX to quit
Knicks
Knicks added.
Enter a team or XXX to quit
XXX
XXX added.
Here are the teams you entered:
Yankees
Mets
Giants
Jets
Knicks
17:12 ~ $ nano loops.py
17:15 ~ $
```

.005 While Loops


If your code is correct your program should behave similar to the example on the left when run in the command line.



```
teams = []
x = ""
while x != "xxx":
    x = ""
    print("Enter a team or xxx to quit")
    x = raw input()
    print (x + " added.")
    if x != "xxx":
        teams.append(x)
print("Here are the teams you entered:")
for team in teams:
    print team
```

.006 Analyzing the Code

In the code example I first declare an empty list called **teams** and then an empty string called **x**.

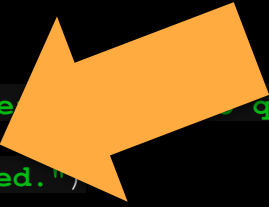


```
teams = []
x = ""
while x != "XXX":
    x = ""
    print("Enter a team or XXX to quit")
    x = raw input()
    print (x + " added.")
    if x != "XXX":
        teams.append(x)
print("Here are the teams you entered:")
for team in teams:
    print team
```

.006 Analyzing the Code

The while loop will continue executing while x is not equal to the string **XXX**.

```
teams = []
x = ""
while x != "xxx":
    x = ""
    print("Enter a team name or quit")
    x = raw_input()
    print(x + " added.")
    if x != "xxx":
        teams.append(x)
print("Here are the teams you entered:")
for team in teams:
    print team
```




.006 Analyzing the Code

The value of `x` is cleared and the user is prompted to enter the name of a team in to the command line.

`raw_input()` allows the user to enter a string in to the command line. In this case, the string is assigned to the variable `x`.

```
teams = []
x = ""
while x != "XXX":
    x = ""
    print("Enter a team or XXX to quit")
    x = raw input()
    print (x + " added")
    if x != "XXX":
        teams.append(x)
print("Here are the teams you entered:")
for team in teams:
    print team
```




.006 Analyzing the Code

If the value of **x** is not **XXX** then the entry is added to the list **teams** using the **append** function.

This will continue until the user enters **XXX** making the while condition false. At that point the loop will exit.



```
teams = []
x = ""
while x != "xxx":
    x = ""
    print("Enter a team or xxx to quit")
    x = raw input()
    print (x + " added.")
    if x != "xxx":
        teams.append(x)
print("Here are the teams entered:")
for team in teams:
    print team
```



.006 Analyzing the Code

When the while loop exits, the list of teams entered, stored in the list **teams** is looped through with the **for** loop, and each team is printed.

```
teams = []
x = ""
while x != "xxx":
    x = ""
    print("Enter a team or xxx to quit")
    x = raw input()
    print (x + " added.")
    if x != "xxx":
        teams.append(x)
print("Here are the teams entered:")
for team in teams:
    print team
```



.007 Activity

Create a program that does the following:

- A) Prints out the numbers between 10 and 20 using a loop.
- B) Prints out the numbers from 100 to 0 counting backwards by 5's using a loop.
- C) Prints out the numbers 1 through 100 but prints an **X** for every other number. (using a loop)
- D) Prints out the numbers 5 - 25 one next to another instead of on separate lines (using a loop).

Good luck.