INFO 450 Spring 2021

Week 4

Coding is Fundamental

Homework Review

Agenda

- Lists
- Sorting
- List Comprehension
- boolean (truthiness)
- conditionals
- if, elif, else

Zen of Python

```
import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
```

- Simple is better than complex.
- Complex is better than complicated.
- Readability counts
- There should be one **and preferably only one** way to do it
- Now is better than never.

Lists

A list is a collection of items in a particular order.

You can put anything you want into a list

Bound by brackets [ ]

Lists should be named something plural e.g. 'students', 'customers', 'deer'

(See what I did there?)

```
import logging
logging.basicConfig(level=logging.DEBUG)

# Create a list of 'bicycle' manufacturers
bicycles = ['trek', 'cannondale', 'redline', 'specialized']

# Get the length (len()) of the list
logging.debug("Length of bicycles: %d", len(bicycles))
logging.debug(bicycles)
```

- I probably won't always enter the import logging and logging.basicConfig calls in each code example, but, you will need to if you use logging.

Printing a List

When you print a list in Python (print or logging, I prefer logging) - Python returns a string representation of your list, brackets included.

```
>>> logging.debug("Length of bicycles: %d", len(bicycles))
DEBUG:root:Length of bicycles: 4
>>> logging.debug(bicycles)
DEBUG:root:['trek', 'cannondale', 'redline', 'specialized']
```

Access Elements in a list

The list contains elements in an order. A developer can access each item by its index:

```
>>> bicycles = ['trek', 'cannondale', 'redline', 'specialized']
>>> logging.debug(bicycles[0])          # Get the first item in the list
DEBUG:root:trek
>>> logging.debug(bicycles[1])          # Get the second item in the list
DEBUG:root:cannondale
>>> logging.debug(bicycles[0].title())    # Call the title function,
                                  # on the first item....
DEBUG:root:Trek
```

Python starts counting/indexing at 0

because it's a real programming language

CRUD

- Create
- Read
- Update
- Delete

```
>>> # Create a list
>>> motorcycles = ['honda','yamaha','harley-davidson']
>>> motorcycles
['honda', 'yamaha', 'harley-davidson']
>>> # Change/update an item in the list
>>> motorcycles[0] = 'ducati'
>>> motorcycles
['ducati', 'yamaha', 'harley-davidson'A
>>> # Append an item to the end (right) of the list
>>> motorcycles.append('honda')
>>> motorcycles
['ducati', 'yamaha', 'harley-davidson', 'honda']
```

- append(itm) - Adds to the end of the list. New, highest index.

Dynamically creating and adding

```
>>> motorcycles = []
>>> motorcycles.append('honda')
>>> motorcycles.append('yamaha')
>>> motorcycles.append('harley-davidson')
>>> motorcycles
['honda', 'yamaha', 'harley-davidson']
```

Inserting , not appending

```
>>> motorcycles
['honda', 'yamaha', 'harley-davidson']
>>> # Insert an item in the middle (index 2) of the list
>>> motorcycles.insert(2, 'suzuki')
>>> motorcycles
['honda', 'yamaha', 'suzuki', 'harley-davidson']
```

Deleting items from the list

Python has a keyword: *del* to delete. This deletes variables from memory, items in the list, etc

```
>>> motorcycles
['honda', 'yamaha', 'suzuki', 'harley-davidson']
>>> del motorcycles[2]
>>> motorcycles
['honda', 'yamaha', 'harley-davidson']
>>> # Get and remove the last (highest index) item
>>> # and store it in a variable
>>> bike = motorcycles.pop()
>>> motorcycles
['honda', 'yamaha']
>>> bike
'harley-davidson'
>>> # Optional parameter to pop is an index
>>> new_bike = motorcycles.pop(0)
>>> motorcycles
['yamaha']
>>> new_bike
'honda'
>>> yamaha_bike = 'yamaha'
>>> # Remove function finds a value that's equal and removes it
>>> motorcycles.remove(yamaha_bike)
```

Organizing a list

- Permanent sort

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> cars
['bmw', 'audi', 'toyota', 'subaru']
>>> # Natural (alphabetical) sort
>>> cars.sort()
>>> cars
['audi', 'bmw', 'subaru', 'toyota']
>>> # Reverse the natural sorting
>>> cars.sort(reverse=True)
>>> cars
['toyota', 'subaru', 'bmw', 'audi']
```

Temporary Sort

Maintain the original ordered list, but provide a sorted 'copy'

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> sorted(cars)
['audi', 'bmw', 'subaru', 'toyota']
>>> sorted_cars = sorted(cars)
>>> sorted_cars
['audi', 'bmw', 'subaru', 'toyota']
>>> cars
['bmw', 'audi', 'toyota', 'subaru']
```

Other things

```
>>> cars = ['bmw', 'audi', 'toyota', 'subaru']
>>> cars
['bmw', 'audi', 'toyota', 'subaru']
>>> cars.reverse()   # Reverse the order of the list
>>> cars
['subaru', 'toyota', 'audi', 'bmw']
>>> len(cars)        # Get the lists length
4
>>> cars[-1]         # Get the last item from the list
'bmw'
>>> cars[0]          # Get the first item in the list
'subaru'
>>> cars[0:1]        # Get a SLICE of the list (notice the type printed)
['subaru']
>>> cars[1:3]        # Get a slice with more than one element
['toyota', 'audi']
>>> cars[55]         # Don't access items that don't exist!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Working with lists

One of the most used control structures in Python has to do with iterating through every element in a list.

Example from the book: in a game, you might want to move every element on the screen.

- Print out todays stock market
- List all orders
- Verify inventory

Iterating lists is probably 75% of what we do in computer programming. Get comfortable.

for

The for loop construct allows you to not worry about how many items are in a list

```
magicians = ['alice', 'pat', 'pallavi', 'sam']

# For loop helps us iterate the list.
# Can be thought of as: "for each item in the list, do something with that item"

# In this example, the individual item will be assigned to the variable: magician
# for each magician in the list of magicians, let's log it
for magician in magicians:
    logging.debug(magician)
logging.debug("DONE")

DEBUG:root:alice
DEBUG:root:pat
DEBUG:root:pallavi
DEBUG:root:sam
DEBUG:root:DONE
```

```
magicians = []
for magician in magicians:
    logging.debug(magician)
```

Lots of errors in the book

Indenting where you shouldn't

Forgetting to indent where you should

Forgetting the colon :

```
for magician in magicians
    logging.debug(magician)
```

Numerical Lists

https://docs.python.org/3/library/functions.html#func-range

The range function generates a series of numbers

class range(stop)
class range(start, stop[, step])
Rather than being a function, range is actually an immutable sequence type,
as documented in Ranges and Sequence Types — list, tuple, range.

```
>>> for x in range(5):
...     logging.debug(x)
...
DEBUG:root:0
DEBUG:root:1
DEBUG:root:2
DEBUG:root:3
DEBUG:root:4
```

```
>>> # range(3,9) effectively returns a list with the values:
>>> # 3,4,5,6,7,8
>>> for x in range(3,9):
...     logging.debug(x)
...
```

.... immutable sequence type?

Notice on the last side I said 'effectively'...

Let's look at some Python fun to see what's going on.

```
>>> x = range(10)
>>> type(x)
<class 'range'>
>>> numbers = list(range(10))
>>> type(numbers)
<class 'list'>
>>> numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List Comprehension

Allows you to perform some actions on a list in one line of code.

Combines the for loop and creation of new elements into one line.

Book: "List comprehensions are not always presented to beginners, but I have included them here because you'll most likely see them as soon as you start looking at other peoples code."

```
>>> # Break it down after
>>> squares = [value ** 2 for value in range(1,11)]
>>> # Surrounded by brackets, so, the end result will be a list.
>>> # for value in range ::  our normal for loop
>>> # value ** 2 == Square each value in the list,
>>> # and the result will be in a new list.
>>> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

cars.py

Example code and output to talk about equality, conditions, etc.

All car manufacturers should be printed 'title' case, but BMW should be all caps.

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:
  if car == 'bmw':
     print(car.upper())
  else:
     print(car.title()
```

```
Audi
BMW
Subaru
Toyota
```

Note: Using print, as those outputs aren't "logging" type statements.

Boolean Expressions

Whether a statement is True or False

Used to track conditions/conditionals

```
>>> game_active = True
>>> can_edit = False
>>> can_edit = true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

Conditionals

"At the heart of every *if* statement is an expression that can be evaluated as True or False and is called a 'conditional test'"

Checking for equality:

```
>>> car = "bmw" # Assigns 'bmw' to variable car
>>> car == "bmw" # Compares variable car to value "bmw"
True
```

```
>>> car = "audi"
>>> car == "bmw"
False
```

Testing strings is 'case sensitive'

```
>>> car = "Audi"
>>> car == "audi"
False
>>> car.lower() == "audi".lower()
True
```

Checking for Inequality

```
>>> requested_topping = "mushrooms"
>>> requested_topping != "anchovies"
True
```

! creates the 'inverse' of the resulting comparison

True becomes False

False becomes True

Numerical Comparisons

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
>>> age < 20
True
>>> age <= 17
False
>>> age > 15
True
>>> age >=18
True
```

```
>>> age = 17
>>> if age < 21:
>>>     print("No drink for you.")

No drink for you.
```

Multiple Conditions

'and' 'or'

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >=21
False
```

AND

	True	False
True	True	False
False	False	False

OR

	True	False
True	True	True
False	True	False

Checking lists

Is it IN the list?

```
>>> requested_toppings = ['mushrooms', 'onions', 'pineapple']
>>> 'mushrooms' in requested_toppings
True
>>> 'bacon' in requested_toppings
False
```

Is it NOT IN the list?

```
>>> banned_users = ['andrew', 'carolina', 'david']
>>> user  = 'marie'
>>> user not in banned_users
True
```

if Statements

'one test and one action, simplest form'

```
if -conditional test-:
    do something
```

```
age = 19
if age >= 18:
    print("You are old enough to vote!")
```

Code sets the age variable, checks it against 18, and then decides to execute the commant.

```
age = 19
if age >= 18:
    print("You are old enough to vote.")
    print("Have you registered to vote?")
```

if-else Statements

if statements help execute an action when conditions are met, but usually programs need to do something else if they aren't met

- If you're 18 or older, you can vote, otherwise, you must wait.

```
age = 17
if age >= 18:
    print("You are old enough to vote.")
    print("Have you registered to vote?")
else:
    print("Sorry, you're too young to vote.")
    print("Please register to vote as soon as you turn 18.")
```

if-elif-else chain

If we need more than one or two 'branches' in the code, we can add more conditional checks

- Under the age of four, tickets are free,
- otherwise if you're under the age of 18, tickets are 25 dollars.
- For everyone else, the cost is $40.

```
age = 12
if age < 4:
    print("Your admission cost is $0.")
elif age < 18:
    print("Your admission cost is $25.")
else:
    print("Your admission cost is $40.")
```

Supports 0 or more 'elif' statements

Once one scenario is reached as True, the rest are ignored.

Else block can be omitted

Testing Multiple Conditions

Sometimes you want more than one action to happen based on events.

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:
    print("Adding mushrooms.")
if 'pepperoni' in requested_toppings:
    print("Oh yes, all the pepperoni.")
if 'extra cheese' in requested_toppings:
    print("Adding extra cheese.")

print("\nFinished making your pizza.")
```

Checking if a list is empty or not

```
requested_toppings = []
if requested_toppings:
    for requested_topping in requested_toppings:
        print(f"Adding {requested_toppings}.")
else:
    print("Are you sure you want a plain pizza?")
```

Using multiple lists

Check our pizza order against an available list of toppings.

```
available_toppings = ['mushrooms', 'extra cheese', 'olives',
                      'pineapple', 'pepperoni', 'green peppers']
requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

for requested_topping in requested_toppings:
    if requested_topping in available_toppings:
        print(f"Adding {requested_topping}")
    else:
        print("Sorry, we don't have {requested_topping}")
```

Styles

```
if age < 4: #looks good with whitespace
    pass # do nothing

if age<4: # looks ugly, add whitespace
    pass
```

Next Week

- Functions
- Dicts
- Tuples
- Input/Output