Lists

Outline

Lists

- This week we'll work with a new type: `list`.
- A `list`, like a tuple, is a sequence of objects
  - Unlike a tuple, a list can grow or shrink
  - We'll see more differences soon
- We write lists with square braces:
  - `[1, 2, 3]`
  - `[]`
  - `["hi", 2, (4, 6)]`

Creating Lists

- Besides square braces (literal lists)...
- We can use `list()` to turn things into lists
  - `list((1, 2))` will create the list `[1,2]` out of the tuple `(1,2)`
  - `list("hi")` will create the list `['h','i']`
  - `list(47)` is an error

Lists as Sequences

Like strings, we can use square braces or slicing to access elements of a list:

```
lst = [1, 2, 3]
three = lst[2]
one_two = lst[0:2]
```

We can also write `len(lst)` to get its length.

Iterating Through Lists

Also like strings, we can iterate through lists with `for ... in ...:`

```
lst = [1, 5, 25, 125]
sum = 0
for number in lst:
    sum += number
```

Exercise: List of Strings

Write a function `net_len(lst)` that takes a list of strings and returns the sum of the lengths of the strings in the list.

Iterating with an Index

We often want to iterate elements and also keep track of an index. But these options can be annoying and error-prone:

```
which = 0
for elt in lst:
    which += 1
    print(f"Element {which} = {elt}")

for which in range(len(lst)):
    elt = lst[which]
    print(f"Element {which} = {elt}")
```

`enumerate` to the Rescue

Instead, let's use the built-in function `enumerate()` that works on any sequence, and throw in some pattern matching for extra niceness:

```
for which, elt in enumerate(lst):
    print(f"Element {which} = {elt}")
```

When you need the index and the element, I'm a big fan of enumerate vs range-loops or tracking an extra variable.

Revenge of the Variables

What will the values of `x` and `y` be after this code executes?

```
x = 5
y = x + 1
x = 10
```

Assignment

Remember: Assignment computes the value on the right and stores it in the location on the left.

Revenge of the Variables

What will the values of `x` and `y` be after this code executes?

```
x = [0]
y = x
y[0] = 10
```

Aliasing

When we assign `x`'s value to `y`, the value is *a list*! That list is an object with a unique identity—it is also a *mutable* object.
When two variables refer to the same object, we say one is an *alias* of the other, or that the object is *aliased* (an alias in English is like an "AKA" name).
Aliasing on its own is fine, but aliasing of a *mutable object* can cause subtle bugs.

Aliasing

Compare:

```
x = [0]
y = x
y[0] = 10
```

```
x = [0]
y = x[:] # slice containing the whole list
y[0] = 10
```

Aliasing

Except for files, lists are the first mutable object we've seen so far. But where a file typically doesn't get passed around between functions much, lists are used as a data structure all over the place.

Copying

Taking a slice of a list creates a new list, so if you don't want the list to change out from under you consider copying it:

```
def start_solitaire(cards):
    deck = cards[:]
    shuffle_deck(deck)
    return deal(deck)
```

Hygiene

When a function receives a list as a parameter, it should document whether:

1. It stores a reference (variable) pointing to this list
2. It may modify this list
3. The caller is allowed to use the list anymore afterwards

When a function returns a list, it should document whether:

1. The caller may modify the list
2. The list may be modified or aliased by some other code

When in doubt, arguments should be copied to be stored and return values should be copied before returning them.

del

We can remove an element from a list using the `del` statement:

```
lst = ["a","b","c"]
del lst[0]
# lst is ["b", "c"]
```

Lists are Objects

Like strings, lists have methods; most of these will modify the list in place.

append

We can use `append` to add an object to the end of a list:

```
lst = []
lst.append("this")
lst.append("is")
lst.append("my")
lst.append("list")
```

What does `lst` contain now?

insert

`insert` is like `append` but works at any position in the list, not just the end:

```
lst = []
lst.append("this")
lst.insert(0, "is")
lst.insert(len(lst), "my")
lst.append("list")
```

What does `lst` contain now?

pop

The inverse of `append` is `pop`, which takes an element out of the list and returns it:

```
lst = [1, 2, 3]
three = lst.pop(2)
lst.append(4)
one = lst.pop(0)
```

What does `lst` contain now?

index

Like `find` for string, we can use `lst.index(elt)` to find the first occurrence of `elt` in the list `lst`.

Modifying the Whole List

We can modify a list in a variety of ways:

- `lst.sort()` will rearrange the elements of `lst` to be in ascending order
- `lst.reverse()` will reverse the order of the elements in the list.
- `lst.clear()` will empty out all the items of the list

Quiz

Assignment 5

Testing As You Go

If you're working on `movie.py`...

- Make a file `test_movie.py`
- `from movie import *`
- Write an assert about a function in `movie.py`
  - Then write the code in `movie.py` that makes it pass

Do the same for `hangman.py` when you get there!

Testing As You Go

This assignment will be really hard if you try to test it all at the end.

Write tests as you go—writing the test first will help you be sure you understand the inputs and outputs of each function.

If you don't know how to write the test, you don't really know what the function ought to do.