ITP 115

Strings

Lists

Sequences Have Indices!

- Each individual item in a sequence is automatically given an position number
- This number is called an **index** and tells what position the item is in
- The **first index** is **zero** (**`0`**)
- The **last index** is the **number of items – 1**

Example: Strings and Indices

```
word = "spamalot"
```

0	1	2	3	4	5	6	7
s	p	a	m	a	l	o	t

- First index is **zero**
- Last index is the **length – 1**
  (8 letters, but last index is 7)

Sequences and Random Access

- Using indices, we can directly access single items from a sequences
- To read a single item from a sequence, we use the [ ] **operator**
- Syntax

```
sequenceVariable[index]
```

Strings – Random Access

0	1	2	3	4	5	6	7
s	p	a	m	a	l	o	t

```
msg = "spamalot"
print(msg[2])
```

```
a
```

```
print(msg[6])
```

```
o
```

Strings – Random Access

0	1	2	3	4	5	6	7
s	p	a	m	a	l	o	t

```
msg = "spamalot"
print(msg[13])
```

```
Error
```

Index Out of Range

- Only valid indices of a sequence are
  `0` to `length-1` *
- Error if you read index beyond `length-1`
  - Also called "Out of bounds"
- **Common mistake**
  - If a sequence has 5 items, what is the index of the last item?

** Python supports negative indices, which go from -1 to -(length). This is not common in programming languages and we won't use it*

Slicing

- We can use `[index]` to get a <u>single item</u> from a sequence
- We can use **slicing** to get <u>multiple items</u> from a sequence
- Slicing works with any sequence (e.g. string, list, etc.)

Slicing

- Syntax

```
sequenceVariable[startPosition:endPosition]
```

Access from start position

*Go **UP TO BUT NOT INCLUDING** end position*

Slicing Strings

0	1	2	3	4	5	6	7
s	p	a	m	a	l	o	t

Examples

```
print(msg[2:6])
```

```
amal
```

```
print(msg[3:4])
```

```
m
```

```
print(msg[0:7])
```

```
spamalo
```

Slicing Strings

0	1	2	3	4	5	6	7
s	p	a	m	a	l	o	t

- What if we want the whole string?

```
print(msg[0:8])
```

```
spamalot
```

Slicing Strings

- What if we want the whole string BUT we don't know how long the string is?

```
msg = input("Enter a word: ")
print(msg[0:len(msg)])
```

*This works because we go from **0** up to but not including **length***

Useful Slicing Tricks

0	1	2	3	4	5	6	7
s	p	a	m	a	l	o	t

- Start at beginning

```
print(msg[:3])
```

```
spa
```

- Go to end

```
print(msg[4:])
```

```
alot
```

- Entire word

```
print(msg[:])
```

```
spamalot
```

Reference only

Note about Slicing

- What is the difference between **a** and **b**?

```
word = "barista"

a = word

b = word[:]

print(a)
                              barista
print(b)
                              barista
```

Reference only

Note about Slicing

- What is the difference between `a` and `b`?

```
word = "barista"

a = word

b = word[:]
```

- *This creates means* **`a`** *is linked to* **`word`**

- *This creates means* **`b`** *is NOT linked to* **`word`**
- *This creates means* **`b`** *is a copy of* **`word`**

We will revisit this later

Two Categories of Sequences

- Mutable – changeable
  - Can modify A SINGLE item in the sequence

- Immutable – unchangeable
  - Can **NOT** modify A SINGLE item in the sequence

Strings are Immutable

```
word = "game"
print (word)
word[0] = "l"
```

```
TypeError: 'str' object does not
support item assignment
```

Strings are Immutable

- Well that's frustrating…
- What kind of sequence is mutable then?

Consider…

Ask the user for three test scores. Display the average along with the original scores.

Create a ***count*** (*set to 0*) and create a ***sum*** (*set to 0*)

```
Ask user for 1st number (store in testScore1)
        Add number to sum and increment counter
Ask user for 2nd number (store in testScore2)
        Add number to sum and increment counter
Ask user for 3rd number (store in testScore3)
        Add number to sum and increment counter
```

Display ***testScore1***, ***testScore2***, ***testScore3***, and average (***sum/count***)

Consider…

Now you have 6 test scores…

Create a ***count*** (*set to 0*) and create a ***sum*** (*set to 0*)

```
Ask user for 1st number (store in testScore1)
        Add number to sum and increment counter
Ask user for 2nd number (store in testScore2)
        Add number to sum and increment counter
Ask user for 3rd number (store in testScore3)
        Add number to sum and increment counter
Ask user for 4th number (store in testScore4)
        Add number to sum and increment counter
Ask user for 5th number (store in testScore5)
        Add number to sum and increment counter
Ask user for 6th number (store in testScore6)
        Add number to sum and increment counter
```

Display ***testScore1***, ***testScore2***, ***testScore3***, ***testScore4***, ***testScore5***, ***testScore6*** and average (***sum/count***)

Consider…

- Using a separate variable for each score…
  - Is impractical for more than a few scores
  - Makes it difficult to use a for loop for efficiency
  - Prone to errors

- All the scores are *related* so….
  - Instead we use a sequence (or group) of variables called a **list**

Lists

- New type of variable!
- Are sequences like strings, but lists are **mutable**
- Contain all the same type of elements*
  - i.e. all strings or all ints

**Technically, Python allows lists to hold different types of elements. For our class, though, we will only store "like items"*

Lists

- Syntax
  `listVariable = [item1, item2, ...]`

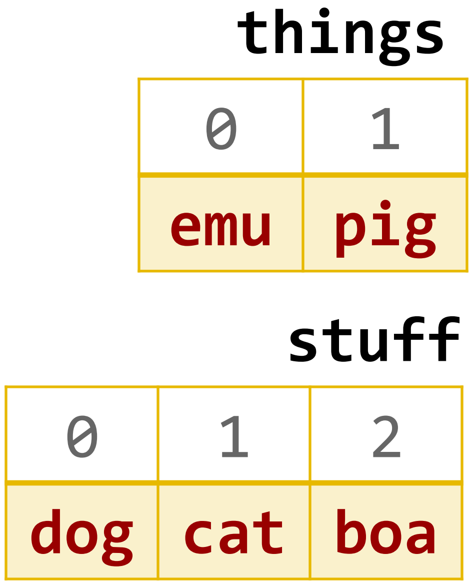
- `item1` could be any type of variable
  - string: `"hello"`
  - int: `7`
  - float: `8.5`
  - another list: `["this is", "another list"]`
  - Any other variable type we will cover

Lists are Sequences

- Since lists are sequences, you can manipulate them just like strings!

Lists are Sequences

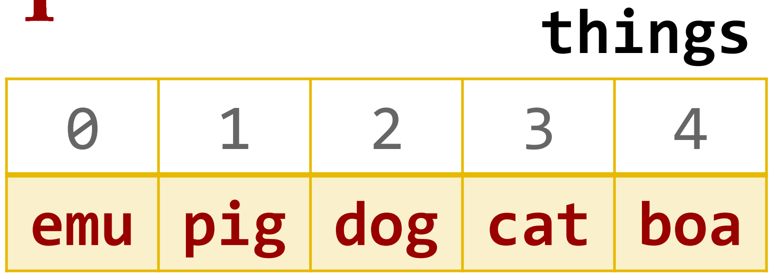
```
things = ["emu", "pig"]
stuff = ["dog", "cat", "boa"]
```



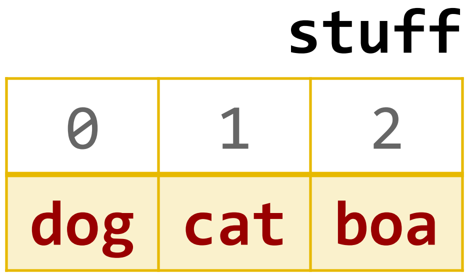
Lists are Sequences

```
things = ["emu", "pig"]
stuff = ["dog", "cat", "boa"]

#concatenate
things += stuff
#alternatively
things = things + stuff
```



0	1	2	3	4
emu	pig	dog	cat	boa



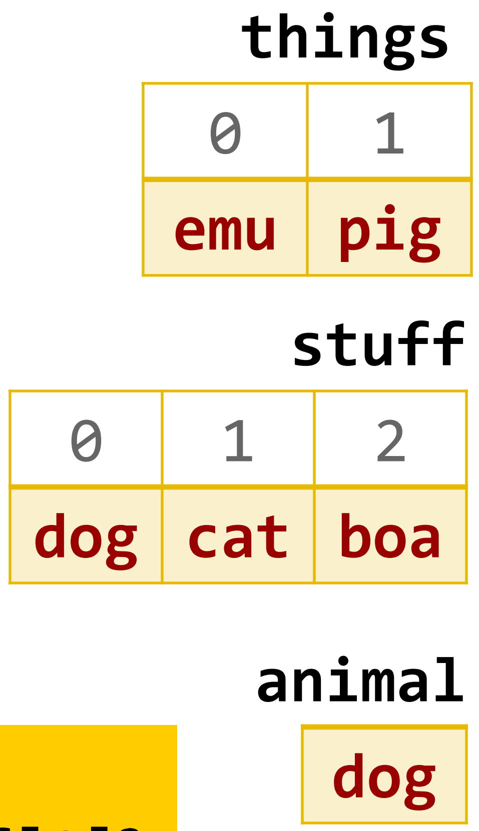
0	1	2
dog	cat	boa

Lists are Sequences

```python
things = ["emu", "pig"]
stuff = ["dog", "cat", "boa"]

#index operator

animal = stuff[0]
```



- What type of variable is **`stuff`**?
- What type of variable is stored at **`stuff[0]`**?
- What type of variable is stored in **`animal`**?

Lists are Sequences

```
things = ["emu", "pig"]
stuff = ["dog", "cat", "boa"]

#slices

grabBag = stuff[0:2]
```

What type of variable is **grabBag**?

things

0	1
emu	pig

stuff

0	1	2
dog	cat	boa

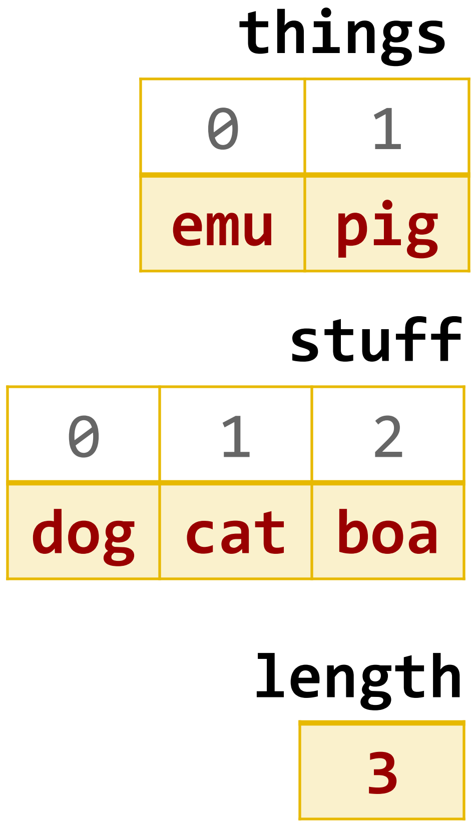
grabBag

0	1
dog	cat

Lists are Sequences

```
things = ["emu", "pig"]
stuff = ["dog", "cat", "boa"]

#len operator
length = len(stuff)
```



Lists are Sequences

```
things = ["emu", "pig"]
stuff = ["dog", "cat", "boa"]

#in operator
if "dog" in stuff:
    print("Found dog")
else:
    print("No dog found")
```

things

0	1
emu	pig

stuff

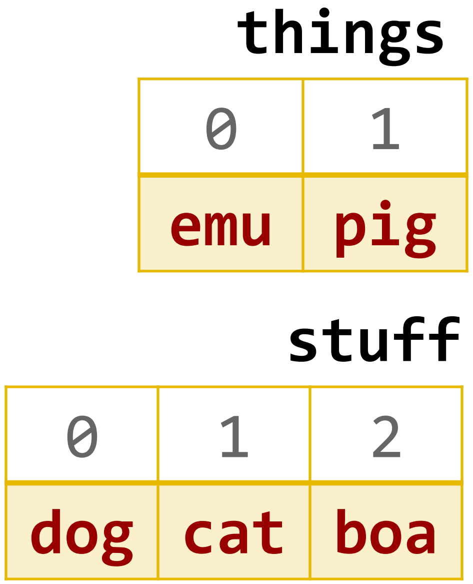
0	1	2
dog	cat	boa

```
Found dog
```

Lists are Sequences

```
things = ["emu", pig]
stuff = ["dog", "cat", "boa"]

#for loop
for item in stuff:
    print(item)
```



```
dog
cat
boa
```

`someList.append(someValue)`

- Adds value to end of a list

- Example

```
numbers = [3, 5, -12]
```

0	1	2
3	5	-12

someList.append(someValue)

- Adds value to end of a list

- Example

```
numbers = [3, 5, -12]
numbers.append(40)
```

0	1	2	3
3	5	-12	40

End Lecture

Creating Empty Lists

- Often we will want to create an empty list before a loop, at the start of our program, etc.

- Syntax

```
numbers = list()
```

or

```
numbers = []
```

Lists are Mutable!

- Assign a new list element by index
- Assign a new list slice
  - Replace multiple items with one item
- Delete a list element
  - Doesn't create a gap in a sequence
  - All the elements "slide down" one position
- Delete a list slice
  - Delete multiple elements

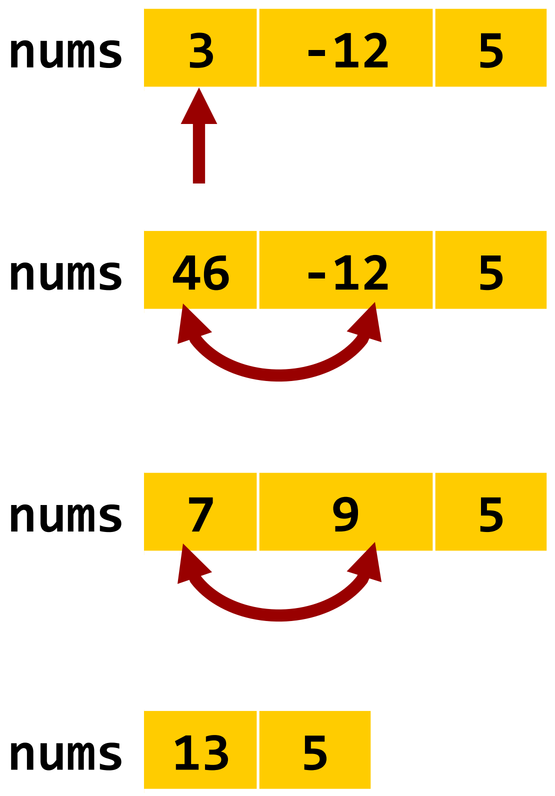
Lists are Mutable!

```
nums        =  [3, -12, 5]

nums[0]     =  46

nums[0:2]   =  [7,9]

nums[0:2]   =  [13]
```



Slice assignment requires value on right to be a list

List Methods

Method	Description
`someList.append(value)`	Adds value to end of a list.
`someList.sort()`	Sorts the elements, smallest value first.
`someList.reverse()`	Reverses the order of a list.
`someList.count(value)`	Returns the number of occurrences of value.
`someList.index(value)`	Returns the first position number of where value occurs.
`someList.insert(i, value)`	Inserts value at position i.
`someList.pop([i])`	Returns value at position i and removes value from the list. Providing the position number i is optional. Without it, the last element in the list is removed and returned.
`someList.remove(value)`	Removes the first occurrence of value from the list.
`del someList[i]`	Removes the element at the specified index

`someList.sort()`

- Sorts the elements, smallest value first
  - Sorts the actual list—**it does NOT return a new list**
- Example

```
numbers = [3, 5, -12, 40]
```

0	1	2	3
3	5	-12	40

someList.sort()

- Sorts the elements, smallest value first
  - Sorts the actual list—**it does NOT return a new list**
- Example

```
numbers = [3, 5, -12, 40]
numbers.sort()
```

0	1	2	3
-12	3	5	40

`someList.sort()`

- Sorts the elements, smallest value first
  - Sorts the actual list—**it does NOT return a new list**
- Example

```
letters = ["b", "a", "g"]
```

0	1	2
b	a	g

someList.sort()

- Sorts the elements, smallest value first
  - Sorts the actual list—**it does NOT return a new list**
- Example

```
letters = ["b", "a", "g"]
letters.sort()
```

0	1	2
a	b	g

`someList.remove(someValue)`

- Removes the first occurrence of a *value* from list
- Example

```
numbers = [3, 5, -12, 40, 5]
```

0	1	2	3	4
3	5	-12	40	5

someList.remove(someValue)

- Removes the first occurrence of a *value* from list
- Example

```
numbers = [3, 5, -12, 40, 5]
numbers.remove(5)
```

0	1	2	3
3	-12	40	5

someList.remove(someValue)

- Removes the first occurrence of a *value* from list
- Example

```
numbers = [3, 5, -12, 40, 5]
numbers.remove(5)
numbers.remove(5)
```

0	1	2
3	-12	40

someList.remove(someValue)

- Removes the first occurrence of a *value* from list
- Example

```
numbers = [3, 5, -12, 40, 5]
numbers.remove(5)
numbers.remove(5)
numbers.remove(5)
```

```
ValueError: list.remove(x): x not in list
```

0	1	2
3	-12	40

Important: Always check **if** value is in list before removing it

`del someList[index]`

- Removes the element from list at *index*

- Example

```
numbers = [3, 5, -12, 40, 5]
```

0	1	2	3	4
3	5	-12	40	5

`del someList[index]`

- Removes the element from list at *index*

- Example

```
numbers = [3, 5, -12, 40, 5]
del numbers[2]
```

0	1	2	3
3	5	40	5

`del someList[index]`

- Removes the element from list at *index*

- Example

```
numbers = [3, 5, -12, 40, 5]
del numbers[2]
del numbers[2]
```

0	1	2
3	5	5

Lists and Strings

- `list()` method
  - Convert a string to a list of characters
- `split()` method
  - Convert words in a string list elements
  - Optional argument (***delimiter***) that specifies which character to use as word boundaries
- `join()` method
  - Convert a list of strings (as elements) into a full string
  - This is a *string* method so you have to invoke it on the delimiter and pass the list as the parameter

`newList = list(someString)`

- Returns a list of all characters in **`someString`**

- Example

```
word = "chihuahua"
letterList = list(word)
print(word)
                 "chihuahua"

print(letterList)
                 ['c', 'h', 'i', 'h', 'u', 'a', 'h', 'u', 'a']
```

`newString = delimiter.join(someList)`

- Returns a string by combining elements in the list
- Elements are separated by the **delimiter**
- **Delimiter** can be any string
- `join` is associated with strings (aka string method), not with lists

```
 someString.join( … )
```

newString = delimiter.join(someList)

- Example

```
wordList = ["Always", "look", "on", "the",
                 "bright", "side", "of", "life"]
delimiter = " "  # means separator between characters
quote = delimiter.join(wordList)
print(quote)
```

```
            Always look on the bright side of life
```

`newList = someString.split(delimiter)`

- Returns a list by separating string everywhere there is a **delimiter** in the string

- **Delimiter** can be any string
  - Common delimiter are be `" "` or `","`

- Ex:`"Ron Weasley,Gryffindor,Red hair"`

Delimiters

newList = someString.split(delimiter)

- Example

```
quote = "spam-spam-spam"
delimiter = "-"
wordList = quote.split(delimiter)
print(wordList)
                    ['spam', 'spam', 'spam']
```

split() vs. strip()

```
line = "Hello my name is Inigo Montoya\n\n    "

aString = line.strip()
print(aString)
                              "Hello my name is Rob"
aList = line.split()
print(aList)
                 ["Hello", "my", "name", "is", "Rob"]
```

Tuples

- Tuples are sequences like lists, but tuples are **immutable**
  - You can NOT change a value in a tuple once it is created

- Tuples behave similarly to lists
  - Tuples can contain elements of any type

Reference Only

Tuples

- Syntax

  `tupleVariable = (item1, item2, ...)`

- `item1` could be any type of variable
  - string: `"hello"`
  - int: `7`
  - float: `8.5`
  - List: `["this is", "another list"]`
  - Any other variable type we will cover

Reference Only

Example

```python
# create an empty tuple
food = ()

# treat the tuple as a condition
if not food:
    print("You don't have any food.")

# create a tuple with some items
food = ("chocolate", "milk", "bread",
"eggs")

# print the tuple
print("The tuple food is: ", food)

# print each element in the tuple
print("Your food items:")
for item in food:
    print(item)
```

```
You don't have any food.

The tuple food is:
 ('chocolate', 'milk', 'bread', 'eggs')

Your food items:
chocolate
milk
bread
eggs
```

Reference Only

Tuples as Sequence

- Since tuples are sequences, you can manipulate them like strings and lists

- Example

```
things = ("emu", "pig")
stuff = ("dog", "cat", "boa")

things += stuff                # concatenate
animal = stuff[0]              # index operator
length = len(stuff)            # len operator
if "dog" in stuff:             # in operator
  print("Found Dog")
```

Tuples are Immutable

```
drinks = ("coffee", "latte", "espresso")

drinks[0] = "americano"

    TypeError: 'tuple' object does not support
                                item assignment
```

Why Use Tuples Instead of Lists

- Tuples are faster than lists
- Tuples' immutability makes them perfect for creating constants since they can't change
- Using tuples can add a level of safety and clarity to your code
- Sometimes tuples are required
  - In some cases, Python requires immutable values