KIT100 PROGRAMMING PREPARATION

Lecture Seven:

Working with Strings and Lists

Lecture Objectives

- Working with Strings
  - Individual and ranges of characters
  - Index
  - Selecting, slicing, and dicing strings
  - Locating a value in strings
  - Searching and formatting strings

- Creating Lists
- Manipulating Lists

Working with Strings

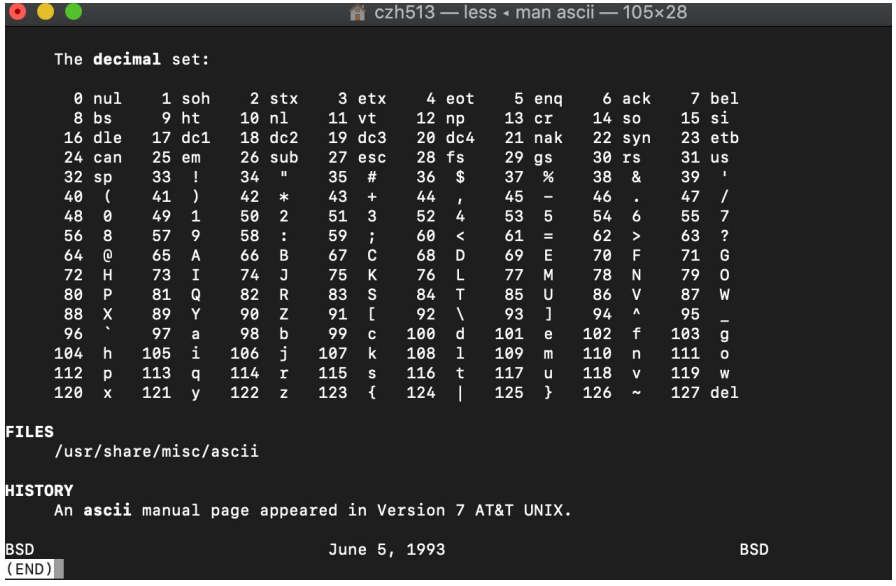
- Remember that computers fundamentally don't understand **strings** (textual data)
  - Computers only deal with **binary data** stored in memory locations.

- We understand strings very well and use them all the time, so programming languages need to provide a way for us to work with strings in code in an easy way, while behind the scenes they are converted or stored in a format the computer understands
  - **string -> (convert to) binary**.

Understanding that strings are different!

- It is difficult to truly understand that computers only understand 0's and 1's.
- **String** are actually made up of sequences of **characters**, and the individual characters are actually stored as **numeric values** (which themselves are actually stored in binary format), based on some underlying representation (such as ASCII or unicode).
  - *Remember the `chr()` and `ord()` functions from lecture 5?)*

Show the table in the terminal (Unix)
>>> man ascii



Defining a character using numbers

Reminder:

- In programming languages, a character has a defined relationship with a number.
- Python uses the American Standard Code for Information Interchange or ASCII (see next slide)

How is the string, " Hello world! " stored? (see in two slides time)

ASCII table

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	**65**	**A**	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	12			

Capital-a, 'A', is stored as ASCII code 65. But at the memory level, this is actually stored as 01000001 in binary.

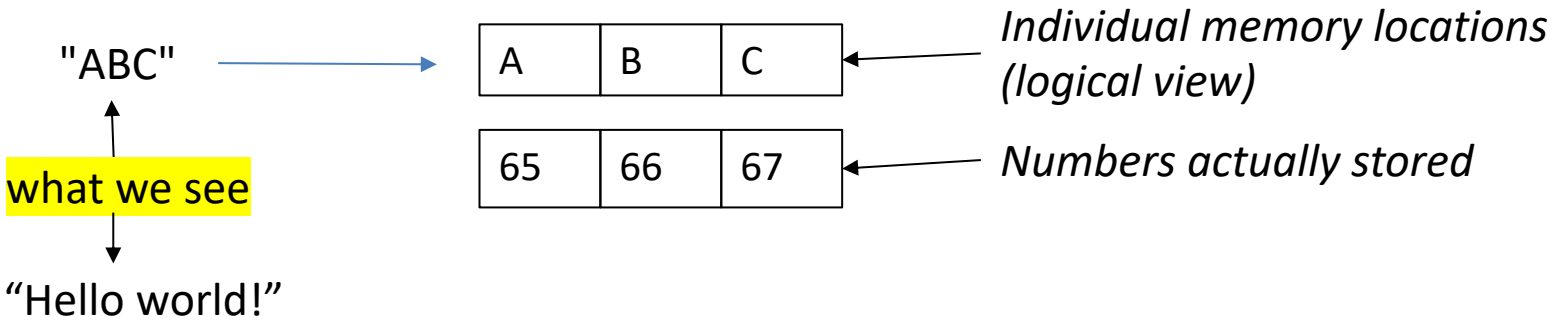
"A" == 65 == 0100001
"a" == 97 == 1100001

```
>>>
>>> ord('A')
65
>>> chr(65)
'A'
>>> bin(ord('A'))
'0b1000001'
>>>
```

```
>>> ord ('a')
97
>>> chr(97)
'a'
>>> bin(ord('a'))
'0b1100001'
>>>
```

Using characters to create strings

- A programming language creates the structure that holds the individual characters together
  - we can manipulate them as a group or individual characters.
- The programming language knows how the structure is designed
  - it's usually many memory locations in a row (with character's number stored in a location) that defines a string.
  - A string variable only refers to the **first** memory location of the **first** character, and the interpreter is clever enough to know more characters follow!



H	e	l	l	o		w	o	r	l	d	!
72	101	108	108	111	32	119	111	114	108	100	33

Is it a ', or a " or a """?

- Remember Python has greater flexibility than most programming languages.

- Working with strings:
  - We know strings can be surrounded by *single quotations* ', or *double quotations* ''.
  - Python will also let us use *triple double quotations* """ to create multi-lined strings.

Is it a ', or a " or a """?

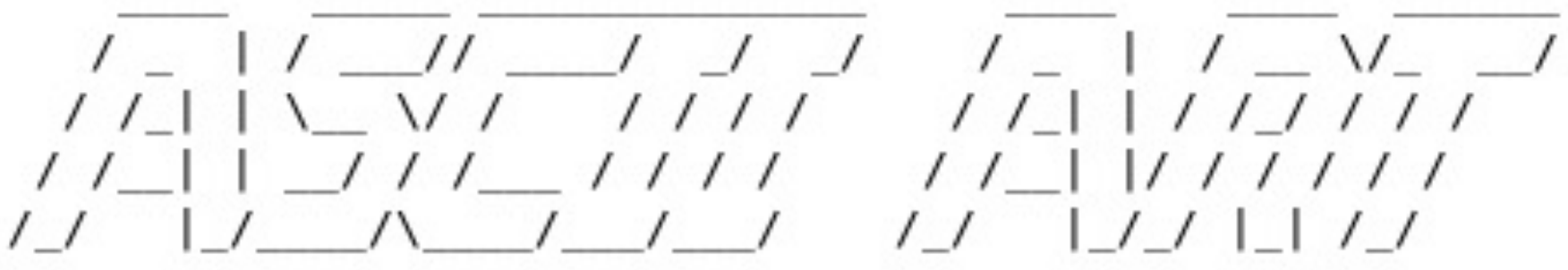

```
"""
Title: Different principles of strings
Purpose: To demonstrate how different types of quotations
marks impact strings
"""

print('Hello there (single quote)!')
print("Hello there (double quote)!")
print("""This is a multiple line
   string using triple double quotes.
   You can also use triple string
   quotes. """)
```

Output

```
Hello there (single quote)!
Hello there (double quote)!
This is a multiple line
   string using triple double quotes.
   You can also use triple string
   quotes.
```

Creating strings with special characters

- Some strings include **special characters**.
  - Control (not visible, but changes how things are displayed)
    - [See the next slide]
  - Accented (e.g. umlaut ü, grave à, etc)
  - Drawing (e.g. primitive drawing)
  - Typographical (formatting indications, e.g. pilcrow ¶)
  - Other



Python control characters

- **Control characters**

  Known as **escape sequences**. There are many different types of escape sequences (see here: https://docs.python.org/3/reference/lexical_analysis.html)

- Most common ones we would use in this unit are :

  `\n` ASCII Line Feed (new line)

  `\r` ASCII Carriage Return (new line)

  `\t` ASCII Horizontal Tab (tab across)

  (`\n` is used more commonly than `\r` but it can be system-dependent)

Python control characters


```
"""
Title: Different principles of strings

Purpose: To demonstrate how different control characters impact strings
"""

print("Part of this text\n is on the next line.")
print("Part of this text\t has been tabbed across.")
```

New line

Tab across

Output:

```
Python 3.8.1 Shell
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week7/w7 le
ure examples/controlChar.py
Part of this text
 is on the next line.
Part of this text        has been tabbed across.
>>>
```

Selecting Individual Characters

- Remember:
  - Python makes it possible to access **individual characters** in a string.

- We can create new strings by *slicing* existing strings and joining existing strings.
  - To **select individual characters** we use the square bracket selector **[ ]**.
    - sometimes call the "**index**" symbol
  - Python strings (as with many languages) are actually immutable, meaning once created, you can't change them.
    - Immutability: for example, you have to create a new string to modify an old string!

Selecting individual characters


```
"""
Title: Different principles of strings

Purpose: To demonstrate how to select individual characters in strings
"""

demoString = "Hello World!"
print(demoString[0])
```

Output:

```
Python 3.8.1 Shell
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week7/w7 lect
ure examples/oneChar_p15.py
H
>>>
```

Selecting individual characters

- Extend individual characters:
  - we can also obtain **a range of characters** from a string.
- We create a range within square brackets, separated by a colon **:** and Python will select the characters in that range.
- We just need to remember that Python uses **zero-based indexing** i.e. the first character is character number **zero**.
- If the first number in the square brackets is omitted, we start from zero. If the second number is omitted, we continue to the end

```
string[start:end] # start through to end-1
string[start:]    # start through the rest of the string
string[:end]      # from the beginning through end-1
string[:]         # the entire string
```

Note we don't include end - we stop one short!

Selecting individual characters


```
"""
Title: Different principles of strings

Purpose: To demonstrate how to select individual characters in strings
"""


firstString = "Hello World"
secondString = "Python is so much fun!"

print(firstString[0])
print(firstString[0:5])
print(firstString[:5])
print(firstString[6:])

thirdString = firstString[:6] + secondString[:6]
print(thirdString)

print(secondString[:7]*5)
```

Note we don't include 5, we stop 4!

Output:

Python 3.8.1 Shell

```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week7/w7 lect
ure examples/charRange_p17.py
H
Hello
Hello
World
Hello Python
Python Python Python Python Python
>>>
```

Slicing and Dicing Strings

- Working with ranges of characters is useful but it doesn't provide us with the capability to **manipulate the string content**.
- **How?**
  - **Slicing and Dicing Strings -** Some of the common functions are:

`capitalize()`	capitalise the first letter of a string
`isalpha()`	returns **true** when the string contains letters only
`isdigit()`	returns **true** when the string contains only digits – no letters
`lower()`	converts all uppercase letters in a string to lowercase letters
`upper()`	converts all lowercase letters in a string to uppercase letters
`len(`*string*`)`	obtains the length of string (how many characters?)

- https://docs.python.org/3.7/library/functions.html

Slicing and Dicing Strings


```
"""
Title: Different principles of strings
Author: Zehong Jimmy Cao Date: Feb 2021
Purpose: To demonstrate how to use various string functions
"""

exampleString = "Hello World"
alphaString = "Hello"
numString = "2021"

print(exampleString.upper())
print(exampleString.lower())
print(exampleString.isalpha())
print(exampleString.isdigit())
print(len(exampleString))
print(alphaString.isalpha())
print(numString.isdigit())
```

Output

```
>>>
= RESTART: /Users/czh513/Desktop/I
gFuncs_p19.py
HELLO WORLD
hello world
False
False
11
True
True
>>>
```

Locating a value in a string

- There are times when we need to locate specific data in a string.

`startswith(prefix [, start[, end]])`	*returns* `True` *if string starts with prefix*
`endswith(suffix [, start[, end]])`	*returns* `True` *if string ends with suffix.*
`find(x, [, start[, end]])`	*determines index where string x occurs, returns -1 if not found*
`replace(old, new [,count])`	*replaces all occurrences of old string with new string and only does* `count` *occurrences if specified*
`count(x [, start[, end]])`	*count how many times string x occurs*

- `[]` *means* ***optional*** *argument* {do not input `[]` }.
- `start` *and* `end` *here are* ***indexes***

  e.g., `print(Utas.startswith("Ut", 0, 2))`
- `start` *and* `end` *can be used to limit where inside a string (i.e. a substring) the search should take place*
  - *if they're not specified, we start at the first letter and go to the last character*

Searching string (1)


```
"""
Title: Different principles of strings
Author: Zehong Jimmy Cao Date: Feb 2020
Purpose: To demonstrate how to use various string search functions
"""

# example 1: Locating a value in a string

test = "UtasICTdiscipline"

print(test.startswith("utas"))
print(test.startswith("Utas"))
print(test.startswith("U", 1)) # return if "U" @index 1
print(test.startswith("U", 0)) # return if "U" @index 0
print(test.startswith("Ut", 0, 1)) # start from 0; the computer do not count "end"
print(test.startswith("Ut", 0, 2)) #start from 0; the computer only count "end-1"

print()
print(test.endswith("line"))

print()
print(test.find("ict"))
print(test.find("ICT"))

print()
print(test.replace("discipline","department"))
print(test.replace("i","k"))
print(test.replace("i","k",1)) # only replace the 1st "i"

print()
print(test.count("i"))
print(test.count("i",10)) # start from index 10
```

Output:

```
>>>
=== RESTART: /Users/czh513/Desktop/KIT001/Teach
False
True
False
True
False
True

True

-1
4

UtasICTdepartment
UtasICTdksckplkne
UtasICTdkscipline

3
2
```

Searching strings (2)


```
# example 2

searchMe = '''Unit KIT001 provides students with an introduction to computer programming and its role in problem solving.
            The ICT graduates of the future require a combination of technical and professional skills
            in order to meet the needs of industry both within Australia and around the world.'''

print()
print("Variable 'searchMe' is: %s" % searchMe)

print()
print("Using the count for 'in':")
print(searchMe.count("in"))
print(searchMe.count(" in ")) # add spaces to avoid the characters in other words.

print()
print("Using the startswith for 'Unit':")
print(searchMe.startswith("Unit"))

print()
print("Using the endswith for 'world:")
print(searchMe.endswith("world"))
print(searchMe.endswith("world.")) # note: the period (.) at the ending point

print()
print("Using the replace:")
print(searchMe.replace("Unit", "Course").replace("KIT001","ICT"))
```

Output:

```
Variable 'searchMe' is: Unit KIT001 provides students with an introduction to computer programming and its role in problem
solving.
            The ICT graduates of the future require a combination of technical and professional skills
            in order to meet the needs of industry both within Australia and around the world.

Using the count for 'in':
8
2

Using the startswith for 'Unit':
True

Using the endswith for 'world:
False
True

Using the replace:
Course ICT provides students with an introduction to computer programming and its role in problem solving.
            The ICT graduates of the future require a combination of technical and professional skills
            in order to meet the needs of industry both within Australia and around the world.
>>>
```

More on formatting Strings!

- Recall: the ***format string %*** operator
  - The form is ***format-string % (comma-separated values)***
    - *e.g., print ("mary %s %d little lambs" % ("had",3))*

- The main emphasis of formatting is to present the string in a form that is both pleasing to the user and easy to understand.

- For example: the user might want a fixed-point number rather than a decimal number as output.

- Commonly use the `format()` function.
  - https://docs.python.org/3.7/library/string.html

Formatting strings

- **type:** Specifies the output type, even if the input type doesn't match. The types are spilt into three groups:
  - *String:* Uses an s or "nothing" at all to specify a string.
  - *Integer:* Uses b (binary); c (character); d (decimal); o (octal); x (hexadecimal with lowercase letters); X (hexadecimal with uppercase letters); and n (locale-sensitive decimal that uses appropriate characters for the thousands separator).
  - *Floating point:* Uses e (exponent lowercase); E (exponent upper case); f (fixed lowercase point); F (fixed uppercase point); g (lowercase general format); G (uppercase general format); n (locale-sensitive decimal that uses appropriate characters for the thousands separator); and % (percentage).

- **align:** Specifies the alignment of data within the display space.
  - **<**: Left aligned
  - **>**: Right aligned
  - **^**: Centered
  - **=**: Justified
  - Recall: we have already used alternate form, e.g. %*width*d

Formatting strings

Examples:

- Previous: print("%s %s" % ("one", "two"))
- New: print("{} {}".format("one", "two"))

- Previous: print("%d %d" % (1, 2))
- New: print("{} {}".format(1, 2))

- Previous: print("%10s" % ('test'))
- New: print("{:>10}".format('test'))

```
>>> print("%s %s" % ("one", "two"))
one two
>>> print("{} {}".format("one", "two"))
one two
>>>
>>> print("%d %d" % (1, 2))
1 2
>>> print("{} {}".format(1, 2))
1 2
>>>
>>> print("%10s" % ('test'))
      test
>>> print("{:>10}".format('test'))
      test
```

Formatting Strings

Another example:

```
"""
Title: Different formatting of strings
Purpose: To demonstrate how to format a string
"""

print("Field formatted as a decimal value:")
formatted = "{:d}" # d (decimal)
print(formatted.format(7000))

print()
print("Field formatted with a thousands separator to output:")
formatted = "{:,d}" # add a thousands separator
print(formatted.format(7000))

print()
print("The {0} of 100 is {1:b}".format("binary", 100)) # b (binary)

print()
print ("My average of this {0} was {1:.2f}%".format("semester", 78.234876)) #f (fixed lowercase point) # keep 2 points (0.23).
```

Output:

```
Field formatted as a decimal value:
7000

Field formatted with a thousands separator to output:
7,000

The binary of 100 is 1100100

My average of this semester was 78.23%
>>>
```

Ln: 59 Col:

Managing lists

- Everyone makes lists in some manner and uses then in various ways to perform an abundance of tasks,
  - e.g. tasks you have to do, items you have to buy, shows you have to watch etc.

- We need to think about how we actually use lists in order to understand how to make Python understand what we've done.

- **Lists** are incredibly **important** in Python.

Organising data in an application

- Python defines a **list** as **an ordered sequence of *things***.
  - Remember Strings? They're a kind of list!

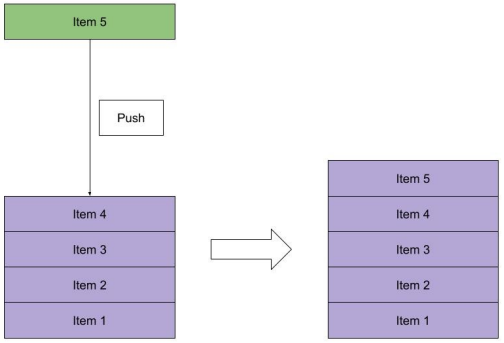
- **Lists**, unlike strings though, **can be changed** (they are mutable) – you can change/add/remove any item in the list.

- **Know more: List is just one approach to organise data. We have more 'data structures' approaches in Python**:
  - Tuples - a sequence of objects (just like a list) except once created they cannot be changed - Tuples are immutable.
  - Dictionaries - Lists let you associate things with numbers, dictionaries let you use anything, not just numbers, to associate one thing with another
  - Stacks – like a stack of books; the first book you place is at the bottom of the stack, to retrieve it you have to remove items above it first (i.e. in reverse order). This is **first-in-last-out (FILO)**
  - Queues - items are added to a queue and the principle of **first-in-first-out (FIFO)** is applied when retrieving item; think of waiting to be served at a shop counter

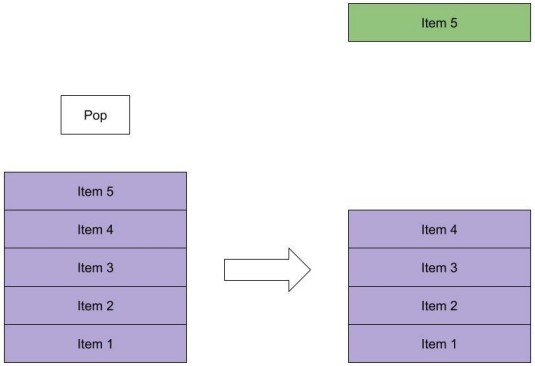
Stacks vs. Queues

Stacks -> first-in-last-out (FILO)

push - adds an element to the top of the stack:

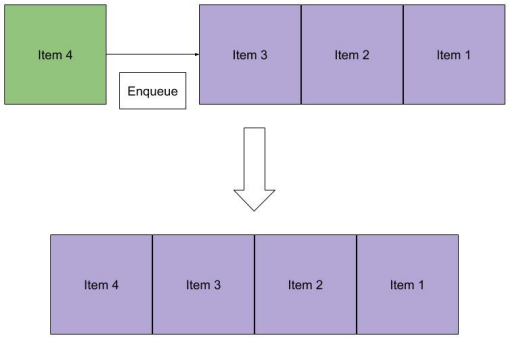


pop - removes the element at the top of the stack:

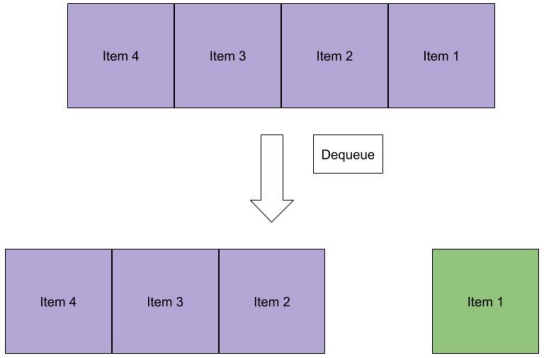


Queues -> first-in-first-out (FIFO)

enqueue - adds an element to the end of the queue:



dequeue - removes the element at the beginning of the queue:



Lists

- **Lists** are the easiest to understand out of all the Python sequences.

- Data is stored in a **list**:
  - it can have any number of items and they may be of different types (integer, float, string etc.).
  - one item comes after another, and we remember the **position** of each item by an **index**.
  - each list **start**s with a particular memory location (which is index 0) and continues from consecutive locations from there.

Creating lists

- A list is finite – meaning there is only a certain number of items in the list.
- Lists are a sequence of values.
- The values are stored in order (with an increasing index)
- An individual value is identified by its position (**index**) in the sequence.

- To define a list,
  - we use square brackets and comma-separate the items
- E.g. `myList = [1,3,5,7]`

Index: `0 1 2 3`

Creating Lists

CreatingLists_p32.py - /Users/czh513/Desktop/CreatingLists_p32.py (3.8.1)

```
"""
Title: Creating Lists Example

Purpose: To provide a demonstration of how to create a list in Python
"""

# example 1

myList = [1,2, "Jimmy"]
print(len(myList))
print(myList[0])
print(myList[2])
print()

print(myList)
myList[0] = "5" # replace
myList[2] = "Jannet"; # replace
print(myList)
print()

# example 2

primes = [2, 3, 5, 7, 11, 13, 17, 19]

print(primes)

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

Output:

```
============== RESTART: /Users/czh513/Desktop/C
3
1
Jimmy

[1, 2, 'Jimmy']
['5', 2, 'Jannet']

[2, 3, 5, 7, 11, 13, 17, 19]
[2, 3, 5]
>>>
```

Accessing Lists


```
primes = [2, 3, 5, 7, 11, 13, 17, 19]

print(primes)

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

The print function knows how to print an entire list!

Output

```
[2, 3, 5, 7, 11, 13, 17, 19]
[2, 3, 5]
```

Can you work out why the output for this is [2, 3, 5] ?

Looping through lists


```
"""
Title: Looping through Lists Example
Author: David Herbert
Date: March 2016
Purpose: To provide a demonstration of how to loop through a list in Python
"""

primes = [2, 3, 5, 7, 11, 13, 17, 19]

for item in primes:
    print(item)
```

Python 3.8.1 Shell

```
Python 3.8.1 (v3.8.1:1b293b6006, Dec 18 2019, 14:08:53)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week7/w7 lect
ure examples/LoopingList_p34.py
2
3
5
7
11
13
17
19
>>>
```

Manipulating lists

- We can **modify** the contents of a list as needed.
- Modifying a list means to change a particular entry, add a new entry, or remove an existing entry.
- **Lists**, unlike strings though, **can be changed** (they are mutable

- Remember C.R.U.D?

Modifying lists

- `list.append(x)` – **Adds** a new entry to the **end** of the list.
- `list.clear()` **or** `del` – **Removes all** entries from the list.
- `newList = list.copy()` – Creates a copy of the current list and places it in a `newList`.
- `list.extend(newlist)` – Appends items from `newlist` onto the **end** of the current list.
- `list.insert(i, x)` – **Adds** a new entry `x` to the position specified by index `i`.
- `list.pop(i)` – **Removes** an entry from the **end** of the list, or from index `i` if it is specified.
- `list.remove(x)` – **Removes** the **first** occurrence of entry `x` from the list if it exists.
- `list.index(x)` – **Returns** the **index** of the **first** item whose value is `x` if it exists.
- `list.count(x)` – **Returns** the **number of times** `x` occurs in the list if it exists.

Modifying lists (1)


```
# example 1 - list functions

myList = ['J', 'i' ,'m', 'm', 'y'] # name: Jimmy
print(myList)
print()

item = myList.pop() # pop(i) - Removes an entry from the end of the list, or from index i if it is specified.
print(myList)
print(item)
print()

item = myList.pop(0)
print(myList)
print(item)
print()

myList.remove('m') # remove(x) - Removes the first occurrence of entry x from the list if it exists.
print(myList)
print()

myList = [1, 2, 3, 4, 5, 6] # index(x) - Returns the index of the first item whose value is x if it exists.
print(myList.index(3))
print()

myList = [1, 1, 1, 1, 1, 2, 2, 1, 2, 3] # count(x) - Returns the number of times x occurs in the list if it exists
print(myList.count(1))
print()

# example 2 - count function

myList = [1, 1, 1, 1, 1, 2, 2, 1, 2, 3]
myList[0] += 1  # index 0 - add "1"
print(myList)

myList[0] += 1 #  index 0 - add "1" again
print(myList)

myList[3] += 2 #  index 3 - add "2"
print(myList)

print()
```

Output:

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 L
ts_p37.py
['J', 'i', 'm', 'm', 'y']

['J', 'i', 'm', 'm']
y

['i', 'm', 'm']
J

['i', 'm']
None

2

6

[2, 1, 1, 1, 1, 2, 2, 1, 2, 3]
[3, 1, 1, 1, 1, 2, 2, 1, 2, 3]
[3, 1, 1, 3, 1, 2, 2, 1, 2, 3]
```

Modifying lists (2) -1

```
"""
Title: Modifying Lists 2
Purpose: To continue the demonstration of how to modify a list in Python
"""

# example 1 - create a list called test_list

testList = []

print("Finding out the length of testList")
print(len(testList))
print()

print("Add a value to testList")
testList.append(1) # append(x) - Adds a new entry to the end of the list.
print()

print("Checking to see if a value has been added to the end of testList.")
print(len(testList))
print()

print("What is the value in position 0 of testList")
print(testList[0])
print()

print("Inserting a value to a particular location.")
#In this case we are inserting value 2 into position 0 of testList
#please note it is the "insert" instead of "replace" function
testList.insert(0,2) # insert(i, x) - Adds a new entry x to the position specified by index i.
print()

print("Displaying our updated testList")
print(testList)
```

Output:

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/wee
ure examples/ModifyingLists_p38_p39.py
Finding out the length of testList
0

Add a value to testList

Checking to see if a value has been added to the end of testList.
1

What is the value in position 0 of testList
1

Inserting a value to a particular location.

Displaying our updated testList
[2, 1]
[2, 1]
```

Modifying lists (2) -2


```
# example 2 - create a list called testList

testList = [2, 1]

print(testList)
print()

print("Creating a copy of testList...")
copyList = testList[:]
print()

# Adding a list to the end of another list
testList.extend(copyList) # extend(newlist) - Appends items from newlist onto the end of the current list
print()

print("Joing testList and copyList creates:")
print(testList)
print()

#Remove a value from the end of a list
testList.pop()
print("Using 'pop()' leaves us with this version of testList")
print(testList)
print()

print("Remove a value from postion 1 in testList with 'remove()'")
testList.remove(1)
print()

print("testList now has these values.")
print(testList)
print()

print("Use 'del.list[:]' to clear the values of testList.")
del testList[:] # or use clear() - removes all entries from the list.
print()

#We can check to see if the list no longer contains values with len()
print("The length of testList is now:")
print(len(testList))
```

Output:

```
Creating a copy of testList...


Joing testList and copyList creates:
[2, 1, 2, 1]

Using 'pop()' leaves us with this version of testList
[2, 1, 2]

Remove a value from postion 1 in testList with 'remove()'

testList now has these values.
[2, 2]

Use 'del.list[:]' to clear the values of testList.

The length of testList is now:
0
>>>
```

Searching lists

- Modifying a list isn't very easy when you don't know what the list contains!
  - We can create a program to **search lists** for specific values.

Searching lists

```
"""
Title: Searching Lists

Purpose: To continue the demonstration of how to search a list in Python
"""

colours = ["red", "orange", "yellow","green", "blue"]

colourSelect = ""

while colourSelect.upper() != "QUIT": # if it equals to "true" -> keep the loop

    colourSelect = input("Please type a colour name, type quit to exit: ")

    if colours.count(colourSelect) >= 1:
        print("The colour exists in the list!")
        print("The colour occurred at index:",colours.index(colourSelect))

    elif colourSelect.upper() != "QUIT":
        print("The list doesn't contain the colour.")
```

Output:

```
Please type a colour name, type quit to exit: red
The colour exists in the list!
The colour occurred at index: 0
Please type a colour name, type quit to exit: green
The colour exists in the list!
The colour occurred at index: 3
Please type a colour name, type quit to exit: silver
The list doesn't contain the colour.
Please type a colour name, type quit to exit: blue
The colour exists in the list!
The colour occurred at index: 4
Please type a colour name, type quit to exit: grey
The list doesn't contain the colour.
Please type a colour name, type quit to exit: quit
>>>
```

Sorting lists

- Reasons for sorting lists is to make it easier for the user to actually see the information in the list, as well as further processing that may require sorted values.
- We can sort an *unsorted* list with Python with the `sort()` method.

Sorting lists


```
"""
Title: Searching Lists

Purpose: To continue the demonstration of how to sort a list in Python
"""
# example - sort
colours = ["red", "orange", "yellow", "green", "blue"]

print("Unsorted colour list: ")
for item in colours:
    print(item, " ", end='') # end='' - avoid creating new lines in 'for' loop
print()

colours.sort() #sort list
print()

print("Sorted colour list: ")
for item in colours:
    print(item, " ", end='')
print()
```

Output:

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teac
ure examples/sortList_p43.py
Unsorted colour list:
red  orange  yellow  green  blue

Sorted colour list:
blue  green  orange  red  yellow
>>>
```

Reverse order sorting

```
# example - reverse sort
print()

colours = ["red", "orange", "yellow", "green", "blue"]

print("Unsorted colour list: ")
for item in colours:
    print(item, " ", end='')
print()

colours.sort() #sort list
colours.reverse() #reversing sort list

print()

print("Reverse sorted colour list: ")

for item in colours:
    print(item, " ", end="")

print()
```

Output:

```
Unsorted colour list:
red  orange  yellow  green  blue

Reverse sorted colour list:
yellow  red  orange  green  blue
>>>
```

Working with the counter object

Counter object

- When we have a short list, we can simply count the values in the list.
  - What happens if we have a really long list?

- The `Counter` object lets us count items quickly.
  - Starting the caption letter C
  - Output format Counter ({values: times})

- It's incredibly easy to use!
- Don't worry if you don't quite follow the example, it's here for future reference if you find a need to use it!

Working with the counter object

```
"""
Title: Using a Counter Objects

Purpose: To demonstrate  how to use a counter object in Python
"""

from collections import Counter # import Counter

myList = [1, 2, 3, 4, 1, 2, 3, 1, 2, 1, 5]

# counter() - track how many times equivalent values are added.
# output format - [value: times]
listCount = Counter(myList) # note: captional letter "C"

print(listCount)

for thisItem in listCount.items():
    print("Item: " , thisItem[0], "Appears:", thisItem[1]) # thisItem[0] = [value]; thisItem[1] = [times]

print("The value 1 appears {0} times.".format(listCount.get(1))) # get(value) - returns the times of the value
```

Output:

```
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week7/w7 lect
ure examples/CounterObjects_p46.py
Counter({1: 4, 2: 3, 3: 2, 4: 1, 5: 1})
Item:  1 Appears: 4
Item:  2 Appears: 3
Item:  3 Appears: 2
Item:  4 Appears: 1
Item:  5 Appears: 1
The value 1 appears 4 times.
>>>
```

Questions?