# CIS 231

Ch. 8 – Python Strings

# 8.1 - Sequences

- Each character in a string has a specific position, or index, in the string.
  - Zero-indexed
  - Indices must be integers

E.g.

```
fruit = "banana"
letter = fruit[1]
print(letter)
a
```

# 8.2 - `len`

- Function that takes a string and returns its length (the number of characters in the string)
- Remember that strings are zero-indexed so the range of indices is 0 to len-1
- Python allows for negative (relative) indices, which start from the back
  - E.g. `fruit[-1]` is the last character, `fruit[-2]` is the second to last character

# 8.3 – Traversal with a `for` loop

- Traversing, or iterating across, a collection, means to visit each location start to finish
- Can do this by starting an index variable at 0 and incrementing until it equals the length
- Python `for` statements also offer `in`, which automagically traverses the string and returns each item one at a time

```
for char in fruit:
    print char
```

# 8.4 – String slices

- A *slice* is a segment of a string that may not necessarily be a single character
- Specified with a *range* of characters inside the square brackets – last index not included

```
# chars from 0 up to but not including 5
s = “Monty Python”
print s[0:5]
Monty
```

- (continued)

# String slices continued

- If you leave out the first index it implicitly starts at the beginning of the string

```
# gets first 3 characters of fruit
fruit[:3]
```

- If you leave out the second index it implicitly goes to the end of the string

```
# gets from the 4th char to the end
fruit[3:]
```

# 8.5 – Strings are immutable

- While you can assign new strings to a variable, you are not allowed to change individual characters (items) of a string in Python
- As an alternative you can create a new string via concatenation

# 8.6 - Searching

- Examine the find() function on p. 74:
  - Iterates through string to find the first occurrence of letter – when found returns index
  - Returns -1 (“not found”) if no match
- Example of a *linear search*
  - Simple but effective
- Alternative approach – next slide

# Searching cont.

- Consider this implementation:

```
def find(word,letter):
    for i in range(len(word)):
        if word[i] == letter:
            return i
    return -1
```

# 8.7 – Looping and counting

- Note if we don't need to worry about what index something happens, but just that it happens, we can easily count occurrences and store them in a counter variable
- `count` variable in example on p. 75
  - Note use of `for-in` loop

# 8.8 – String methods

- Methods are functions that are defined for certain objects (more later this semester)
- Methods are called (*invoked*) in the context of a particular object (strings are objects)
- `.` after the name of the object variable

```
# change string in word to upper case
word.upper()
```

  - Since the object has some/all of the data needed to execute the method, often fewer/no parameters

# 8.9 The `in` operator

- Using `in` with two strings gives a boolean result based on whether (or not) the first operand is a substring of the second
- `in_both()` function on p. 76:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print letter
```

# 8.10 – String comparison

- The relational operators support strings, allowing you do to a *lexical comparison*

```
if word == “banana”:
   print “All right, bananas.”

if word < “banana”:
   print word + “, comes before banana.”
else if word > “banana”:
   print word + “, comes after banana.”
else:
   print “All right, bananas.”
```

# Next Time

- Ch. 9 – Case study: word play
  - Reading word lists
  - Search
  - Looping with indices