Python
Strings

Learning Objectives

- Introduce and discuss strings
- Identify the similarities and differences between strings and lists

Lists and Sequences

- **Recall:** lists are a sequence of values connected by a common name

```
grades = [76, 65, 98]
```

- Lists have many methods to manipulate their contents

```
grades.append(83)
grades.sort()
print(grades)
```

```
[65, 76, 83, 98]
```

- **Methods** are functions attached to some object and are accessed via the . operator

Strings

- Like a list, a string is a sequence of values, specifically a sequence of characters
- Basic string initialization uses quotes, either single ‘’ or double “”

```
fname = "Bryan"
lname = 'Burlingame'
print(fname + ' ' + lname)
```

```
Bryan Burlingame
```

- As with lists, individual letters can be accessed via the [] operator

```
print(fname[0])
```

```
B
```

Remember: the index starts at zero

len

- Many of the tools and techniques used with lists, also work with strings
- len() also works on strings much like it does on lists, returning the

```
grades = [76, 65, 98, 83]
fname = "Bryan"
print("grades:\t" + str(len(grades)))
print("fname:\t" + str(len(fname)))
```

```
grades: 4
fname:  5
```

Traversal

- **Recall**: traversing a collection is the process of working on the collection element by element
- As with lists, either a for loop or a while loop can be used to traverse a string

```
fname = "Bryan"
index = 0
while index < len(fname):
    print(fname[index])
    index += 1
```

```
B
r
y
a
n
```

```
fname = "Bryan"
for letter in fname:
    print(letter)
```

```
B
r
y
a
n
```

Slicing a string

- The slice operator [:] works on strings just as it does on lists
- For string[a:b] return the characters from a to (b-1)
  - If a is omitted, assume 0
  - If b is omitted, assume the end of the string
  - If both are omitted, return a copy of the string

```
name = 'Bryan Burlingame'
print(name[0:5])
print(name[:5])
print(name[6:])
```

```
Bryan
Bryan
Burlingame
```

Mutability

- **Recall:** lists are mutable – the elements of the list can be changed
- Strings are immutable. The characters of a string cannot be changed
  - The “grades[1] = 72” line passes, the error is on the next line

```
grades = [76, 65, 98, 83]
fname = "Bryan"
grades[1] = 72
fname[1] = 'y'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-d7ef576a7e52> in <module>()
      2 fname = "Bryan"
      3 grades[1] = 72
----> 4 fname[1] = 'y'

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

Mutability

- **Recall:** lists are mutable – the elements of the list can be changed
- Strings are immutable. The characters of a string cannot be changed
  - The “grades[1] = 72” line passes, the error is on the next line
  - Use slices and concatenation, instead

```
grades = [76, 65, 98, 83]
fname = "Bryan"
grades[1] = 72
fname = fname[:1] + 'y' + fname[2:]
print(fname)
```

```
Byyan
```

Search

- Search is the process of traversing a sequence, evaluating each member of that sequence for some value, and returning when the value is found
- From the text:

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return(index)
        index += 1
    return -1

fname = "Bryan"
print(find(fname, 'y'))
```

```
2
```

Strings Methods

- Strings have many methods
- Since strings are immutable, many methods return a string
- Note: slices of a string are considered strings as well

```
fname = 'bryan'
print(fname.capitalize())
print(fname.upper())
print(fname.upper().lower())
print(fname.center(20, '*'))
print(fname.find('ya'))
print(fname[0].isupper())
print(fname[1:2].islower())
```

```
Bryan
BRYAN
bryan
*******bryan********
2
False
True
```

https://docs.python.org/3/library/stdtypes.html#string-methods

String Formatting

- Strings are generally intended to be consumed by people. Attractive and consistent formatting makes this easier
- Python has a mini-language just for formatting
  - Takes the form “Format Spec”.format(thing_to_be_formatted)

```
a = 3
b = 40
c = 500
print('{0} {1} {2}'.format(a, b, c))
```

```
3 40 500
```

  - Each element of the format list is matched in order with a replacement field {#}
  - {0} == a, {1} == b, {2} == c, where the format is specified within the {} and between the {} (note the spaces in the format string and the output)

https://docs.python.org/3/library/string.html#formatstrings

String Formatting - Examples

- Note how the values follow the replacement field and the inter-replacement field characters are added

```
a = 3
b = 40
c = 500
print('{0} {1} {2}'.format(a, b, c))
print('{0},{1},{2}'.format(a, b, c))
print('{0},{0},{0}'.format(a, b, c))
print('{2}   ,   {1}   ,   {0}'.format(a, b, c))
```

```
3 40 500
3,40,500
3,3,3
500   ,   40   ,   3
```

String Formatting - Examples

- Formatting of the values are controlled in the replacement field
  - {#:format_spec}
  - Ex: {0:.2f} – format the 0th value, display as a float with 2 decimal places

```
a = 3
b = 40
c = 500
print('{0:.2f},{1:+.2f},{2:.2f}'.format(a, b, c))
print('{0:>8.2f}{1:>8.2f}{2:>8.2f}'.format(a, b, c))
print('{0:^12b}{1:^12b}{2:^12b}'.format(a, b, c))
print('{0:*^10.2f},{1:$^10.2f},{2:#^10.2f}'.format(a, b, c))
```

```
3.00,+40.00,500.00
    3.00   40.00  500.00
     11        101000    111110100
***3.00***,$$40.00$$$,##500.00##
```

String Formatting - General

The general form of a *standard format specifier* is:

```
format_spec     ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill            ::=  <any character>
align           ::=  "<" | ">" | "=" | "^"
sign            ::=  "+" | "-" | " "
width           ::=  digit+
grouping_option ::=  "_" | ","
precision       ::=  digit+
type            ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" |
```

- Each of the options on the format_spec line can be included
- Print a minimally 10 character wide field, field. Center the value. Include the + sign, show as a 2 decimal point float. Fill in any white space with an asterisk *

```
a = 3
print('{0:*^+10.2f}'.format(a))
```

```
**+3.00***
```

https://docs.python.org/3/library/string.html#formatstrings

String Formatting - General

The general form of a *standard format specifier* is:

```
format_spec     ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill            ::=  <any character>
align           ::=  "<" | ">" | "=" | "^"
sign            ::=  "+" | "-" | " "
width           ::=  digit+
grouping_option ::=  "_" | ","
precision       ::=  digit+
type            ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" |
```

- Each of the options on the format_spec line can be included
- Print a minimally 10 character wide field, field. Center the value. Include the + sign, show as a 2 decimal point float. Fill in any white space with an asterisk *

```
a = 3
print('{0:*^+10.2f}'.format(a))
```

```
**+3.00***
```

https://docs.python.org/3/library/string.html#formatstrings

String Formatting - General

The general form of a *standard format specifier* is:

```
format_spec     ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill            ::=  <any character>
align           ::=  "<" | ">" | "=" | "^"
sign            ::=  "+" | "-" | " "
width           ::=  digit+
grouping_option ::=  "_" | ","
precision       ::=  digit+
type            ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" |
```

- All the values in the format_spec are optional
- Go to the docs to understand what each value controls
- Note that this is a very succinct mini-language

```
a = 3
print('{0:*^+10.2f}'.format(a))
```

```
**+3.00***
```

https://docs.python.org/3/library/string.html#formatstrings

String Formatting - General

- The format_spec is simply a string; therefore, it can be transformable

```
a = 3
form = ''
for i in range(10):
    form = '{0:*^' + str(i) +  'd}'
    print(form.format(a))
```

```
3
3
3*
*3*
*3**
**3**
**3***
***3***
***3****
****3****
```

Understanding References(1)

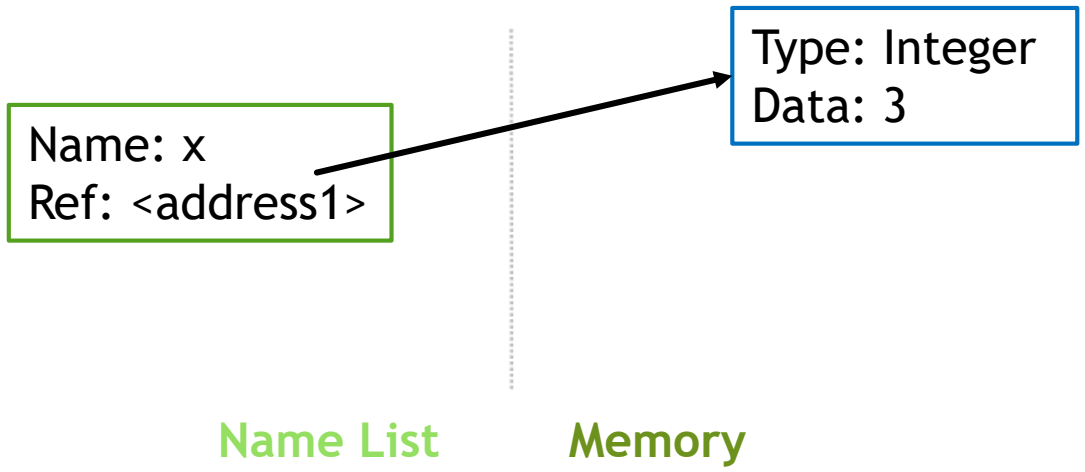
- Assignment manipulates references
  - x = y **does not make a copy** of the object y references
  - x = y makes x **reference** the object y references
- **Very useful; but beware!**
- **Example:**

```
>>> a = [1, 2, 3]   # a now references the list [1, 2, 3]
>>> b = a           # b now references what a references
>>> a.append (4) # this changes the list a references
>>> print b         # if we print what b references,
[1, 2, 3, 4]        # SURPRISE! It has changed…
```

- Why??

Understanding References (2)

- What happens when we write x = 3?
- First, an integer **3** is created and stored in memory
- A name **x** is created
- A **reference** to the memory location storing the **3** is then assigned to the name **x**



Understanding References (3)

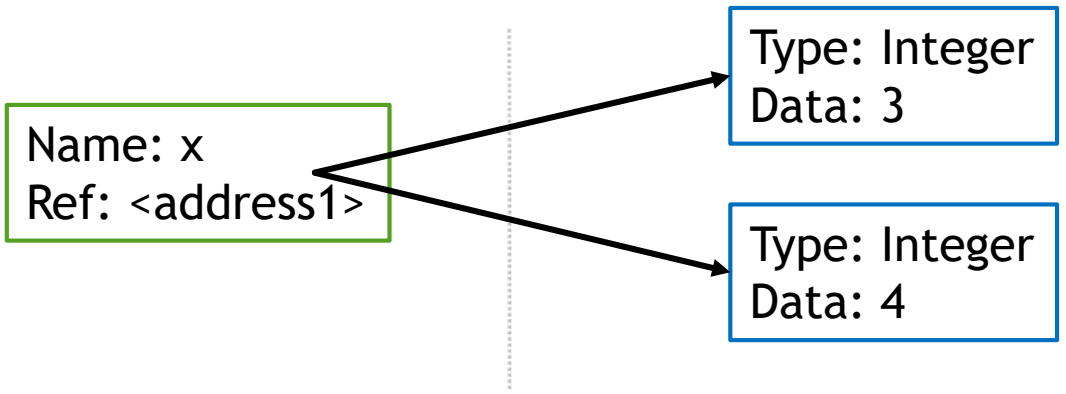
- The data 3 we created is of type integer. In Python, the data types integer, float, and string (and tuple) are "**immutable**."
- This doesn't mean we can't change the value of x, i.e. *Change what x refers to …*
- For example, we could increment x:

```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

Understanding References (4)

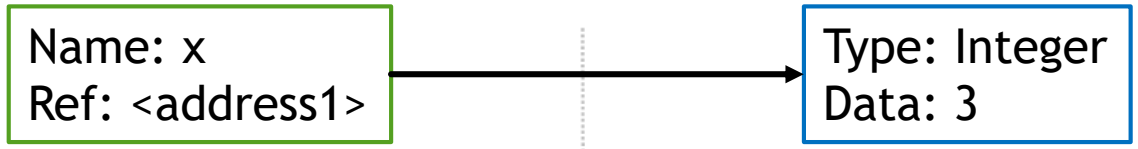
- **If we increment x, then what's really happening is:** **>>>x = x +1**

1. *The reference of name **x** is looked up*
2. The value at that reference is retrieved.
3. The 3+1 calculation occurs, producing a new data element **4** which is assigned to a fresh memory location with a new reference.
4. The name **x** is changed to point to this new reference.
5. *The old data **3** is garbage collected if no name still refers to it.*



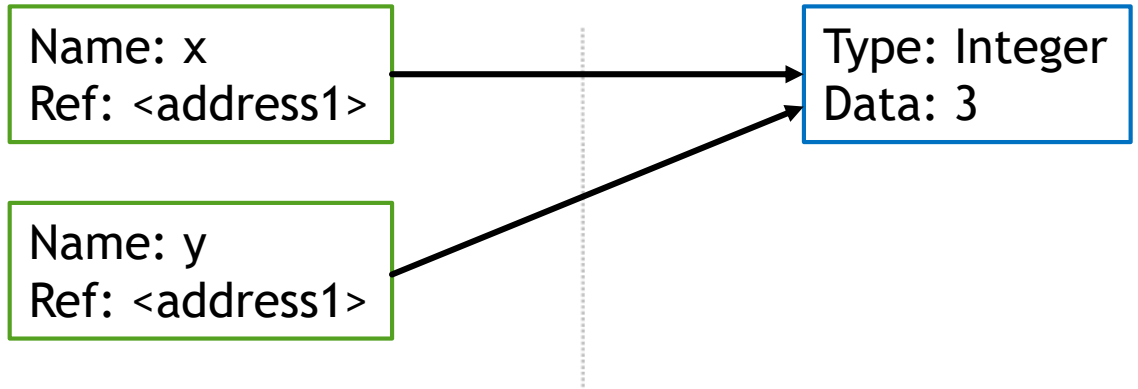
Assignment (1)

```
>>> x = 3     # Creates 3, name x refers to 3
>>> y = x     # Creates name y, refers to 3.
>>> y = 4     # Creates ref for 4. Changes y.
>>> print(x)      # No effect on x, still ref 3.
3
```



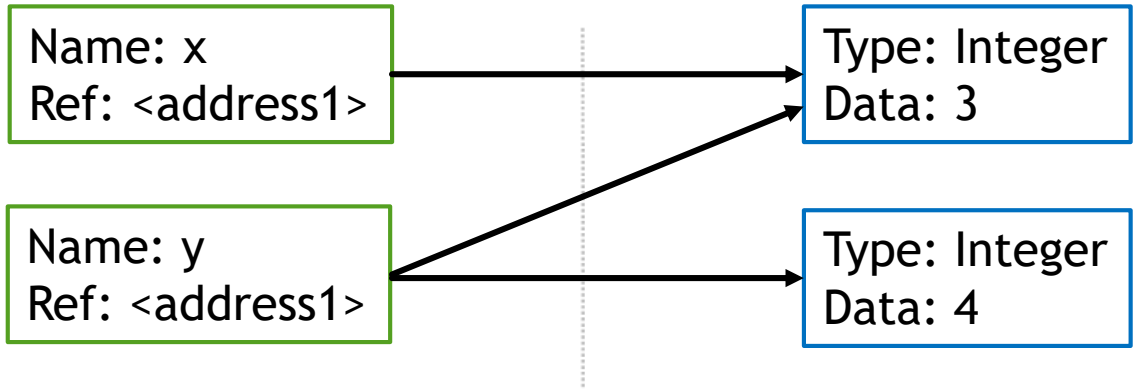
Assignment (1)

```
>>> x = 3     # Creates 3, name x refers to 3
>>> y = x     # Creates name y, refers to 3.
>>> y = 4     # Creates ref for 4. Changes y.
>>> print(x)      # No effect on x, still ref 3.
3
```



Assignment (1)

```
>>> x = 3       # Creates 3, name x refers to 3
>>> y = x       # Creates name y, refers to 3.
>>> y = 4       # Creates ref for 4. Changes y.
>>> print(x) # No effect on x, still ref 3.
3
```



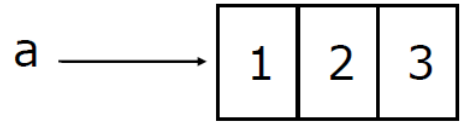
Assignment (2)

- **For other data types (lists, dictionaries, user-defined types), assignment works differently.**
  - These data types are "**mutable**."
  - When we change these data, we do it *in place*.
  - We don't copy them into a new memory address each time.
  - If we type y=x and then modify y, both x and y are changed.

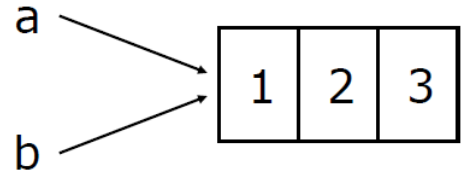
Return to the First Example

```
>>> a = [1, 2, 3]      # a now references the list [1, 2, 3]
>>> b = a              # b now references what a references
>>> a.append(4)        # this changes the list a references
>>> print b            # if we print what b references,
[1, 2, 3, 4]           # SURPRISE! It has changed…
```

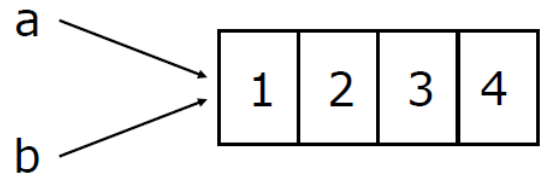
a = [1, 2, 3]



b = a



a.append(4)



Resources

- Bryan Burlingame's notes
- Downey, A. (2016) *Think Python, Second Edition* Sebastopol, CA: O'Reilly Media
- (n.d.). 3.7.0 Documentation. *String Methods — Python 3.7.0 documentation*. Retrieved October 16, 2018, from https://docs.python.org/3/library/stdtypes.html#string-methods