ENGR 298: Engineering Analysis and Decision Making - Lists

Dr. Jason Forsyth

Department of Engineering

James Madison University

A program is a set of instructions for manipulating, storing/storing, and making decisions on data in your computer.

Variables, Collections, and Data Structures

- To date we have stored information in single variables or pulled information from sequences of variables (lists).

- Programming languages provide various data structures to store information. Each data structure provides various improvements for space, access time, sorting...etc.

- Will focus on two built-in data structures: *lists* and *tuples*. Both are considered *sequence* structures.

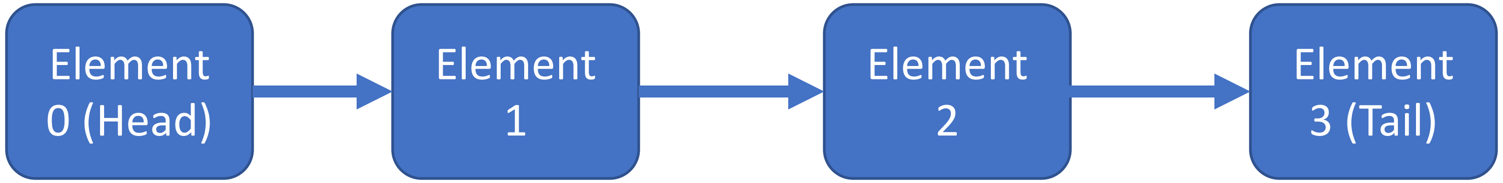
A program is a set of instructions for **manipulating**, **moving/storing**, and **making decisions** on **data** in your computer.

Basic Data Structure: Lists in Python

- A list is a ***sequence of objects*** stored in a data structure. Python lists are ***mutable*** where elements can be added, removed, sorted…etc.

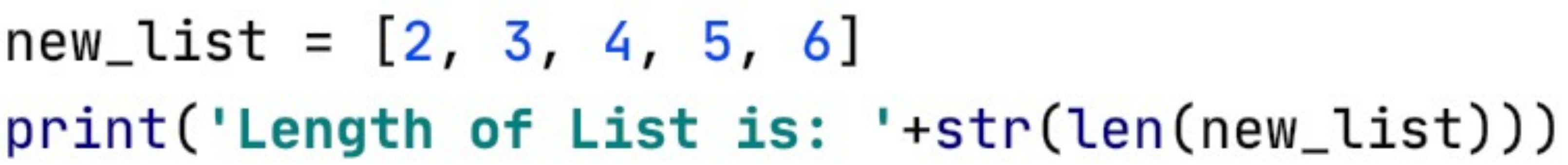
- Lists are *typically* **homogenous** containing the same type of information; however, this is not a requirement in Python.

- Lists are ***iterable***, so they can be used in for loops (last week). The first element in the list is the *head*, the end is the *tail*.

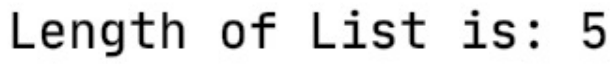


Creating Lists in Python – Part 1

- Lists can be declared with [….] to manually enumerate the elements.

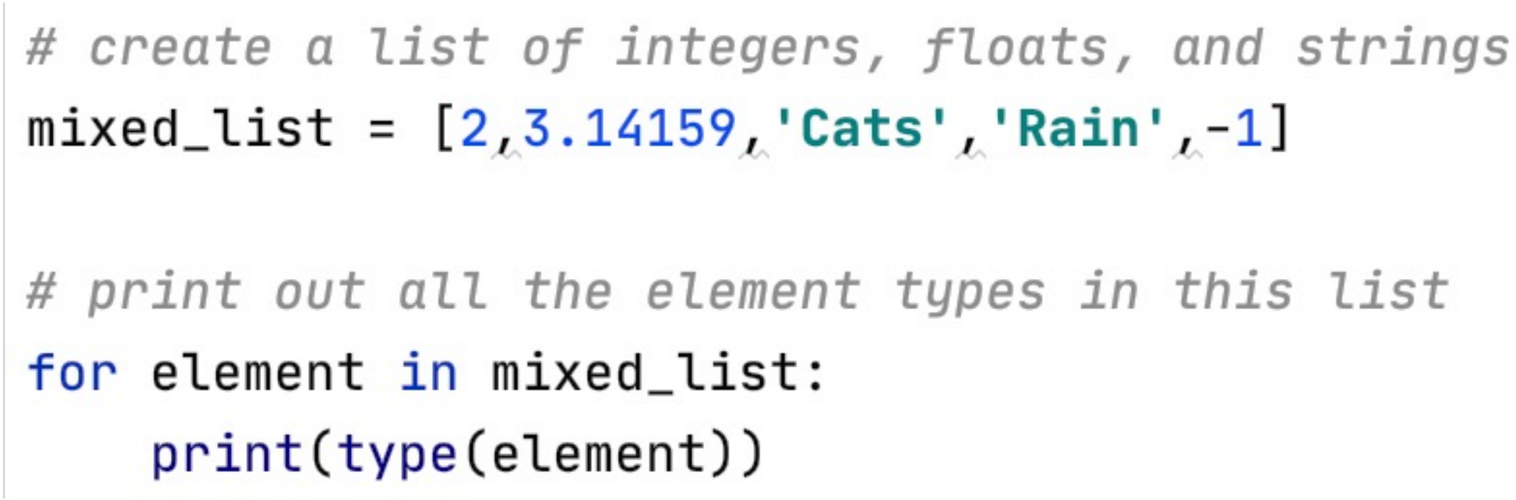


```
new_list = [2, 3, 4, 5, 6]
print('Length of List is: '+str(len(new_list)))
```

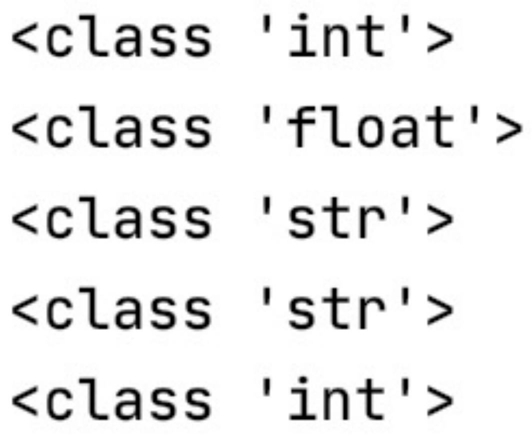


```
Length of List is: 5
```

- Lists are iterable so it is easy to traverse their elements.



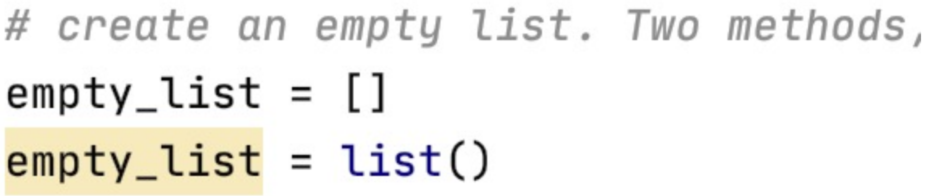
```
# create a list of integers, floats, and strings
mixed_list = [2,3.14159,'Cats','Rain',-1]

# print out all the element types in this list
for element in mixed_list:
    print(type(element))
```



```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'str'>
<class 'int'>
```

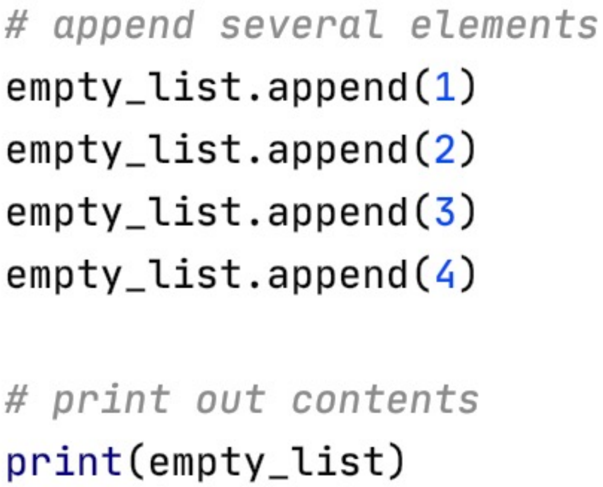
Creating Lists in Python – Part 2

- An empty list can be created with [] or using the **List constructor** list()

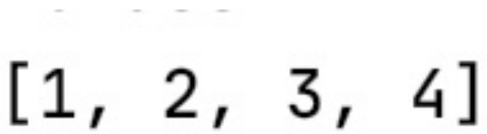


```
# create an empty list. Two methods,
empty_list = []
empty_list = list()
```

- New elements can be added by **appending** to the end of the list



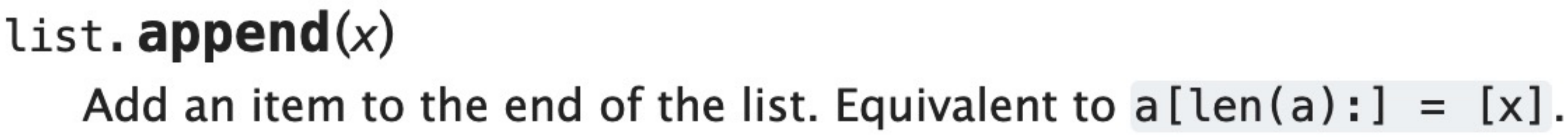
```
# append several elements
empty_list.append(1)
empty_list.append(2)
empty_list.append(3)
empty_list.append(4)

# print out contents
print(empty_list)
```



```
[1, 2, 3, 4]
```

Creating Lists in Python – Part 3

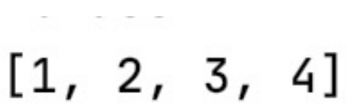
- Very important that append() treats the passed argument as a singular object.



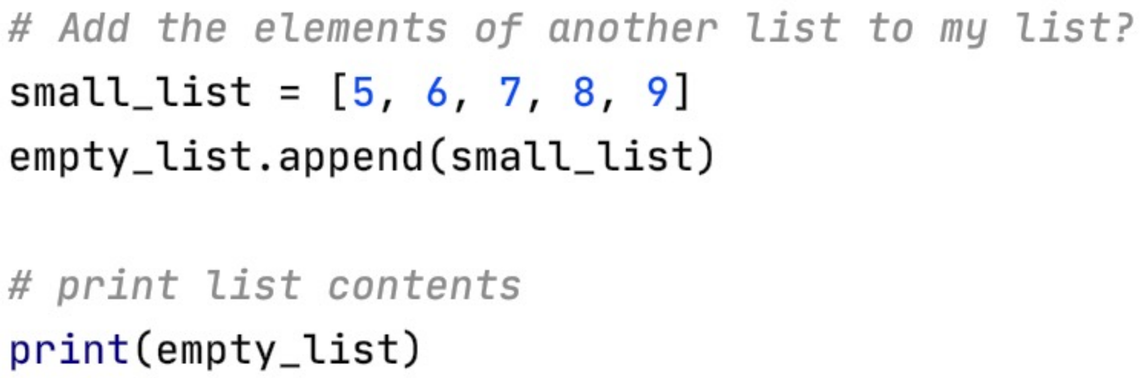
list.**append**(*x*)

Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

- What do you expect the following code to do:



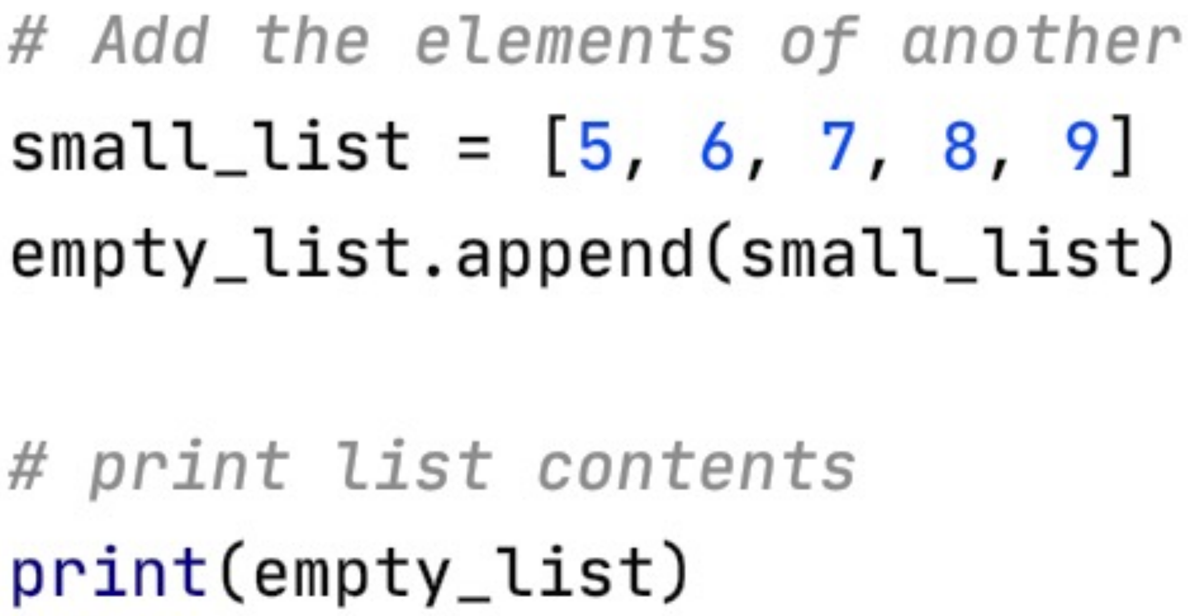
```
[1, 2, 3, 4]
```



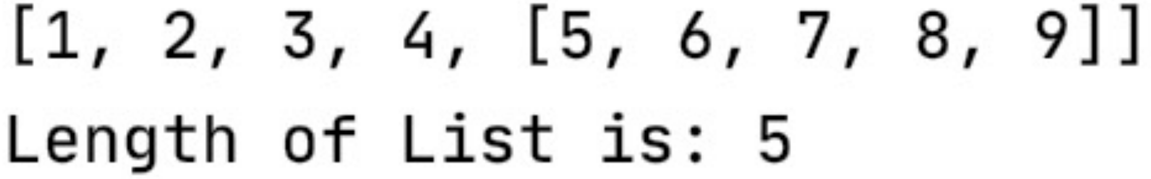
```
# Add the elements of another list to my list?
small_list = [5, 6, 7, 8, 9]
empty_list.append(small_list)

# print list contents
print(empty_list)
```

Creating Lists in Python – Part 3

- Append treated *small_list* as a single object. New list now has 5 elements, not 9. Lists contain objects. A list is also an object.



```
# Add the elements of another list to my list?
small_list = [5, 6, 7, 8, 9]
empty_list.append(small_list)

# print list contents
print(empty_list)
```

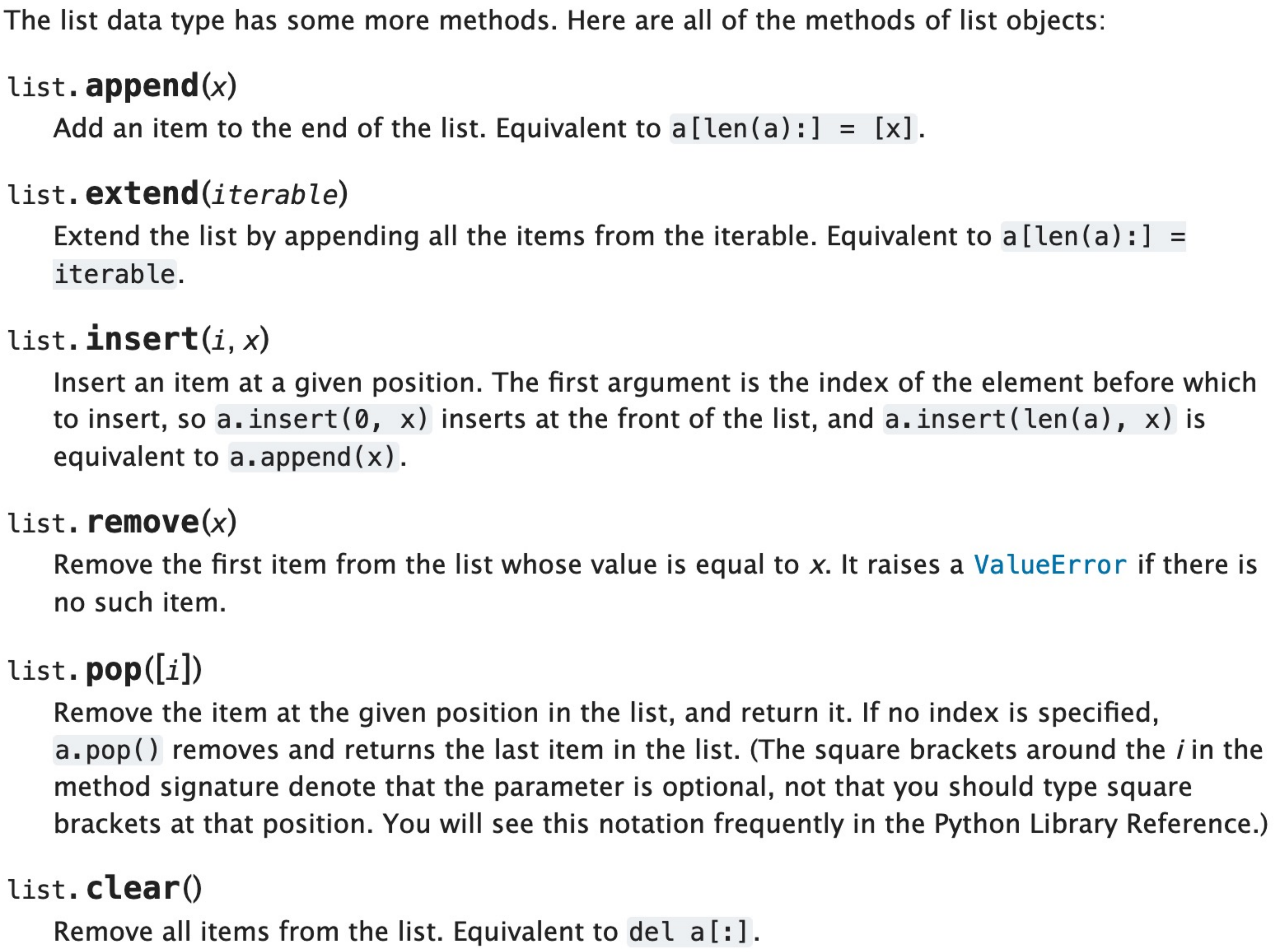


```
[1, 2, 3, 4, [5, 6, 7, 8, 9]]
Length of List is: 5
```

If we wanted to add elements of a list, into another list, what method should be used?

What if we wanted to insert elements into the middle of the list?

Note the difference between remove() and pop().



The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**(*x*)

Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**(*`iterable`*)

Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

`list.`**`insert`**(*i*, *x*)

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**(*x*)

Remove the first item from the list whose value is equal to *x*. It raises a `ValueError` if there is no such item.

`list.`**`pop`**([*i*])

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

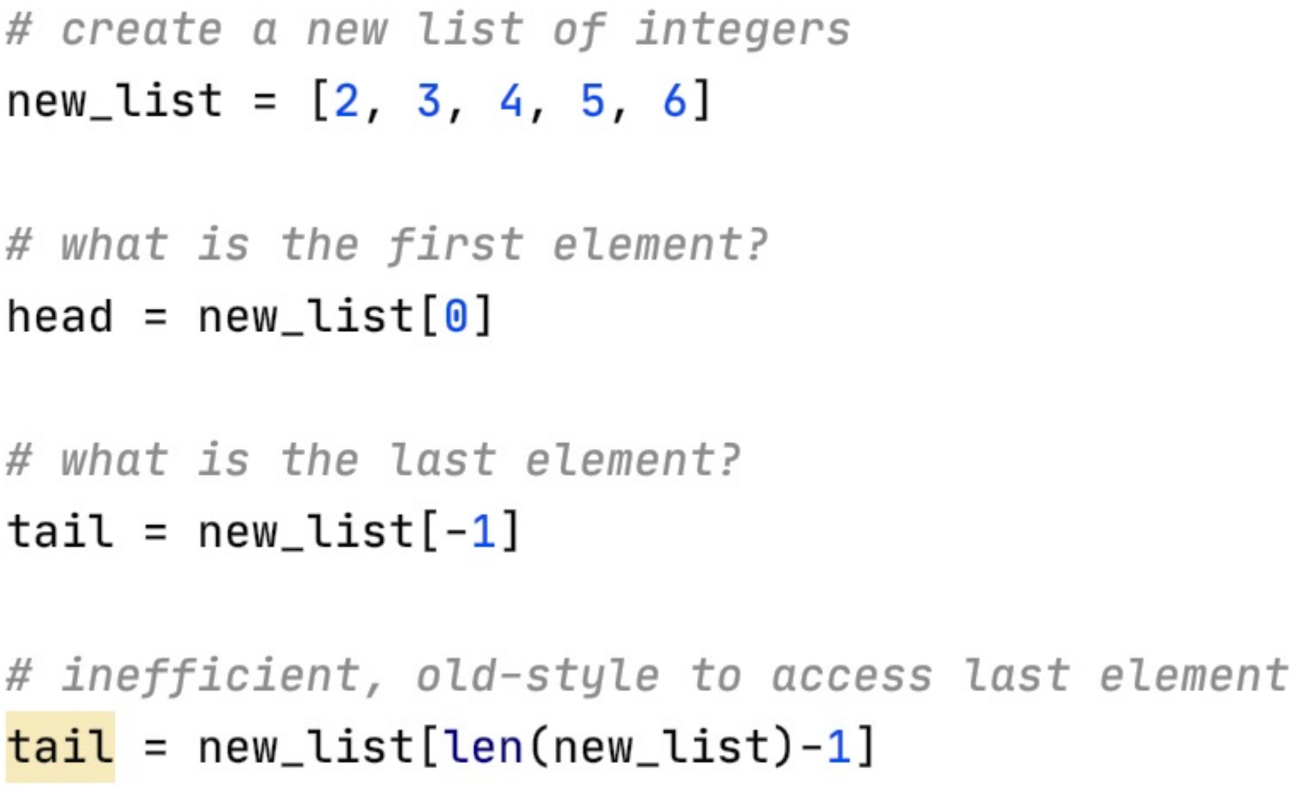
`list.`**`clear`**()

Remove all items from the list. Equivalent to `del a[:]`.

Accessing Elements in a List: Slicing, Dicing, and Such…

Accessing List Contents

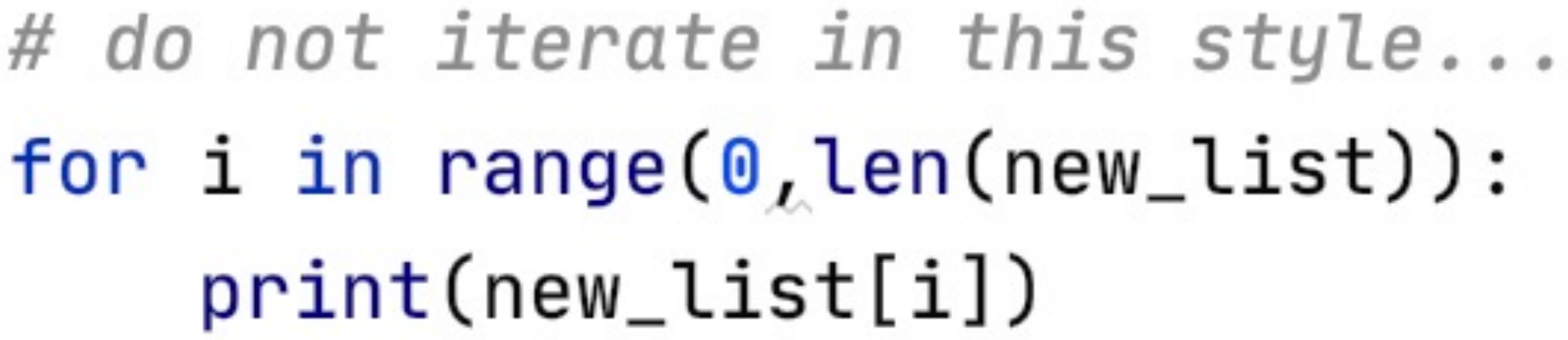
- Can access list element with array-list arguments *[idx]*, where *idx* is the index position of the element in the list [0,length).



```python
# create a new list of integers
new_list = [2, 3, 4, 5, 6]

# what is the first element?
head = new_list[0]

# what is the last element?
tail = new_list[-1]

# inefficient, old-style to access last element
tail = new_list[len(new_list)-1]
```

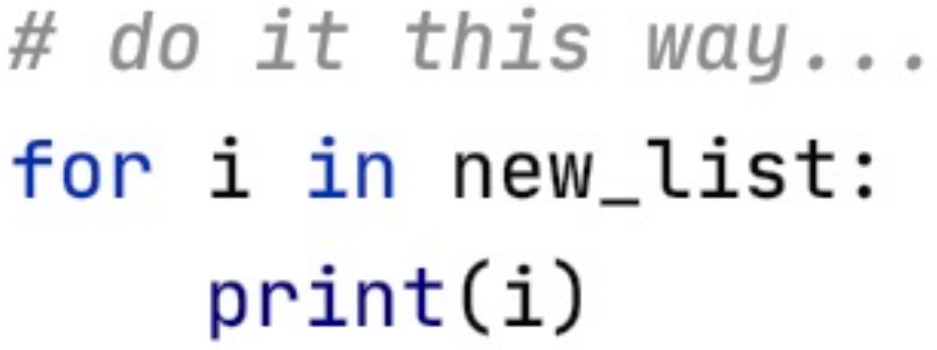
Remember, list is iterable

No need for index variable to iterate across list. Just use iterator ability in Python.



```
# do not iterate in this style...
for i in range(0,len(new_list)):
    print(new_list[i])
```

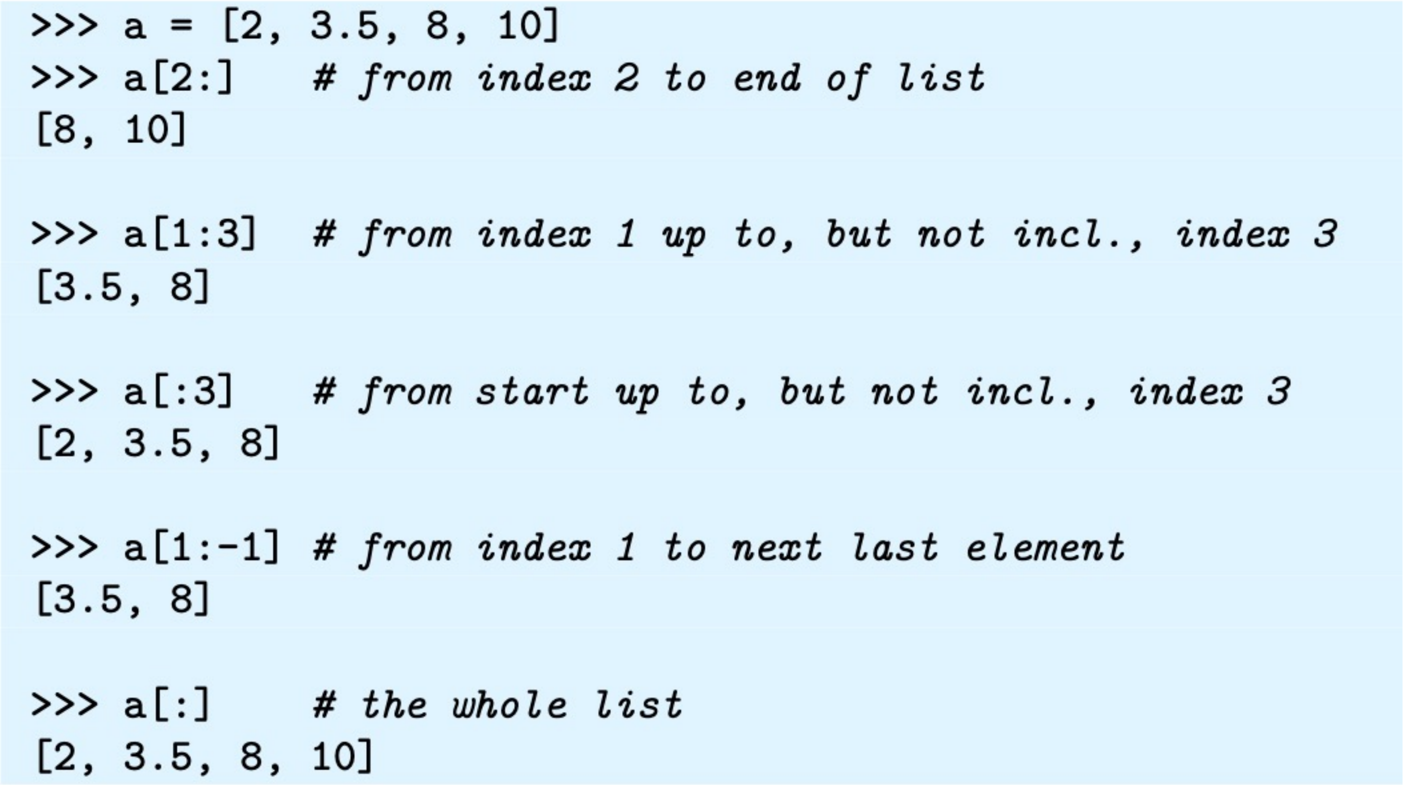
There may be times the index variable style is required. However, it will be much slower access times.



```
# do it this way...
for i in new_list:
    print(i)
```

List Slicing

- List elements can be accessed in many patterns to create new sublists. This slicing should be familiar from MATLAB.



```
>>> a = [2, 3.5, 8, 10]
>>> a[2:]    # from index 2 to end of list
[8, 10]

>>> a[1:3]   # from index 1 up to, but not incl., index 3
[3.5, 8]

>>> a[:3]    # from start up to, but not incl., index 3
[2, 3.5, 8]

>>> a[1:-1] # from index 1 to next last element
[3.5, 8]

>>> a[:]     # the whole list
[2, 3.5, 8, 10]
```

Differences Between Lists and Arrays

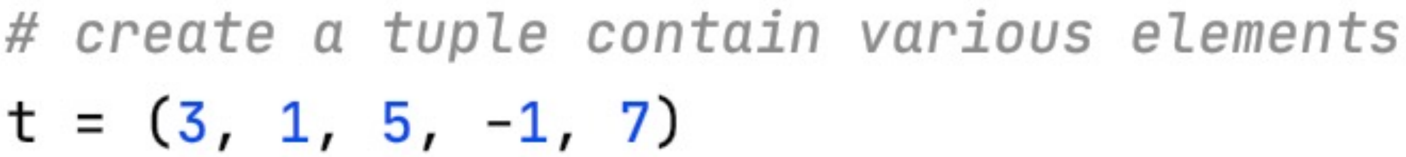
- Lists and arrays have common access patterns [] but there are important differences.

- A list is *accessed* like an array, but it is a single dimension sequence of objects. List objects can be added, removed, modified. Are unlikely to be ‘near’ in memory. Objects can be various types.

- An array is a multi-dimension block of information. Elements can be modified but are unlikely to be added/removed. Objects are generally the same time and considered to be a ‘local’ in memory.

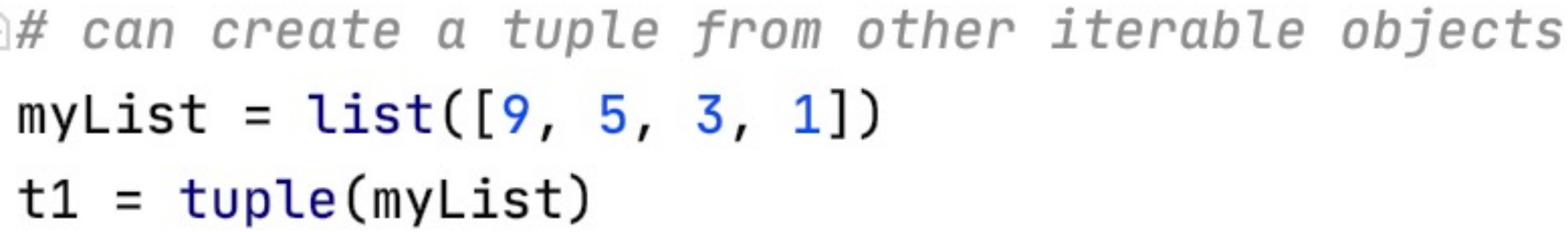
Tuples

- A Tuple is like a List in that it can contain many objects of different types. Tuple is **iterable**. However, elements are **immutable**. Tuple cannot be sorted.



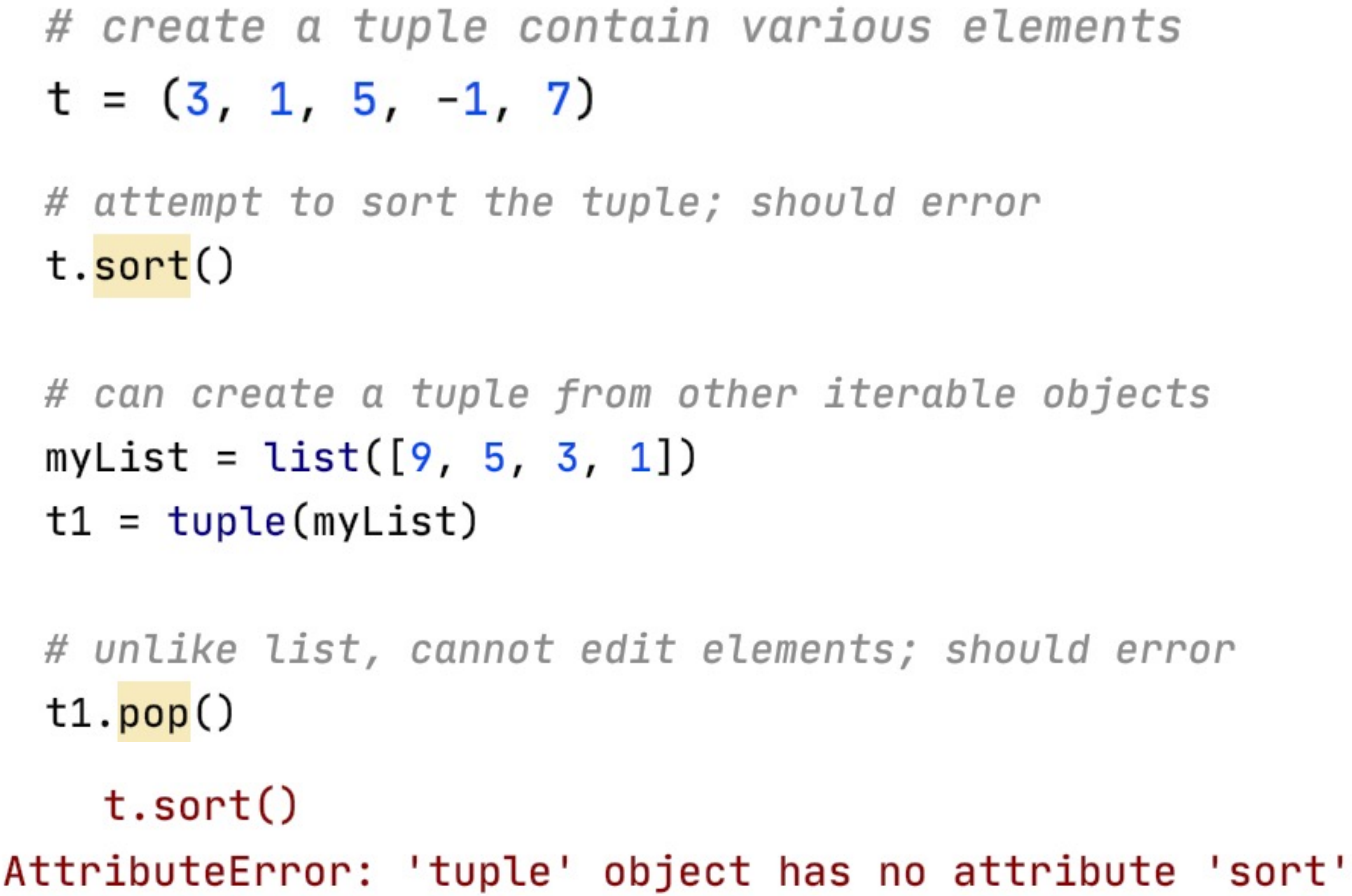
```
# create a tuple contain various elements
t = (3, 1, 5, -1, 7)
```

- Tuple is created with (…) or tuple() if passed an iterable.



```
# can create a tuple from other iterable objects
myList = list([9, 5, 3, 1])
t1 = tuple(myList)
```

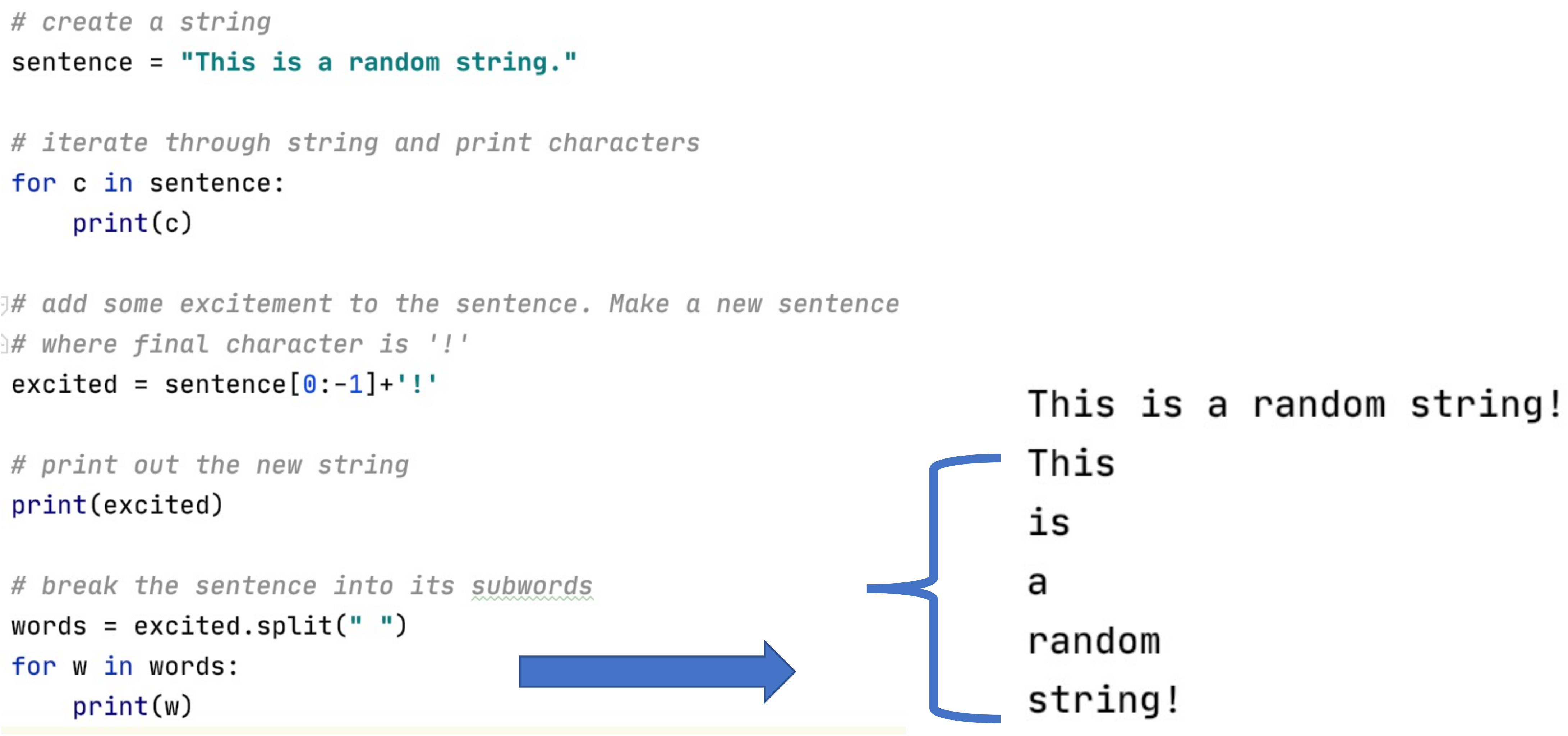
Tuples are Immutable



```
# create a tuple contain various elements
t = (3, 1, 5, -1, 7)

# attempt to sort the tuple; should error
t.sort()


# can create a tuple from other iterable objects
myList = list([9, 5, 3, 1])
t1 = tuple(myList)


# unlike list, cannot edit elements; should error
t1.pop()
```

```
    t.sort()
AttributeError: 'tuple' object has no attribute 'sort'
```

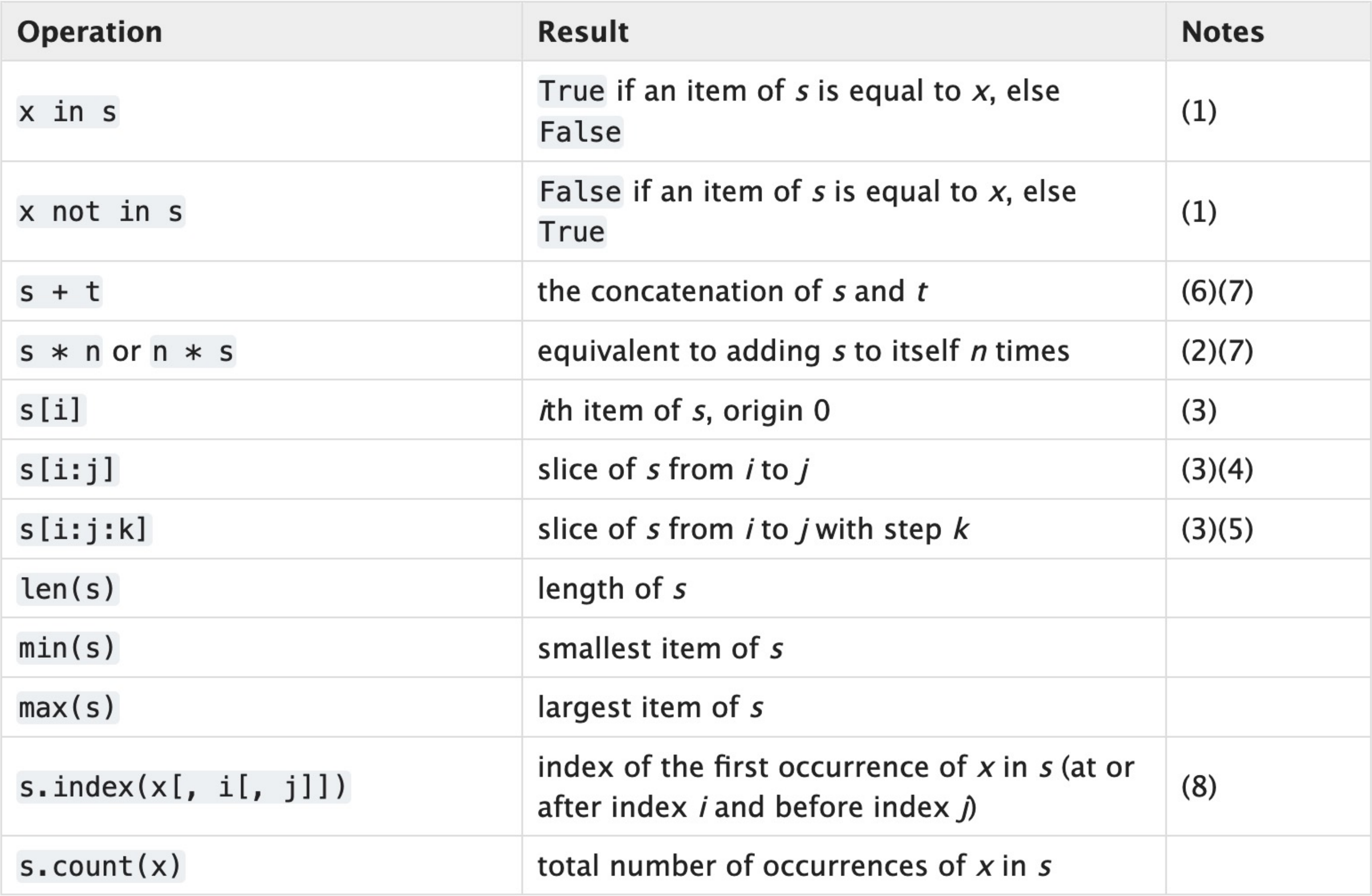
Stings are Immutable Lists of Characters



```python
# create a string
sentence = "This is a random string."

# iterate through string and print characters
for c in sentence:
    print(c)

# add some excitement to the sentence. Make a new sentence
# where final character is '!'
excited = sentence[0:-1]+'!'

# print out the new string
print(excited)

# break the sentence into its subwords
words = excited.split(" ")
for w in words:
    print(w)
```

```
This is a random string!
This
is
a
random
string!
```

Lists and Tuples are “Sequences”



Operation	Result	Notes
`x in s`	`True` if an item of *s* is equal to *x*, else `False`	(1)
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`	(1)
`s + t`	the concatenation of *s* and *t*	(6)(7)
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times	(2)(7)
`s[i]`	*i*th item of *s*, origin 0	(3)
`s[i:j]`	slice of *s* from *i* to *j*	(3)(4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3)(5)
`len(s)`	length of *s*	
`min(s)`	smallest item of *s*	
`max(s)`	largest item of *s*	
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)	(8)
`s.count(x)`	total number of occurrences of *x* in *s*	

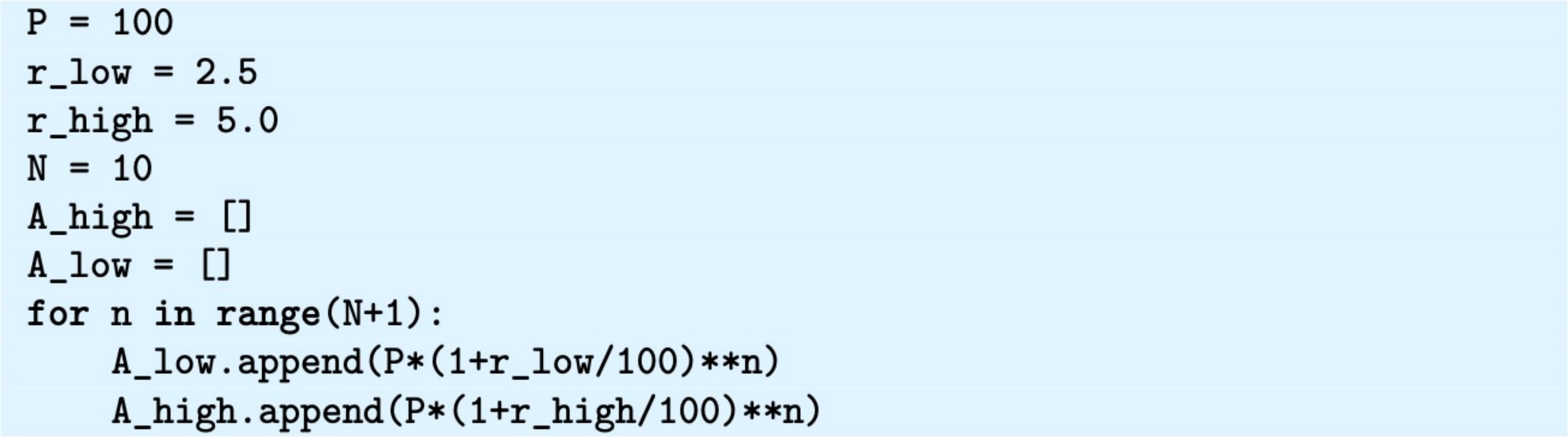
Week 3 Tasks. Use templates on GitHub

- **Odds and Evens**: Given a list *nums*, place all even numbers in a list called *evens*, and vice versa for the *odd* values. Use template on GitHub.

- **Vectors**: Perform the dot product of two vectors where each vector is stored as a list of values. Note: will need to install numpy.

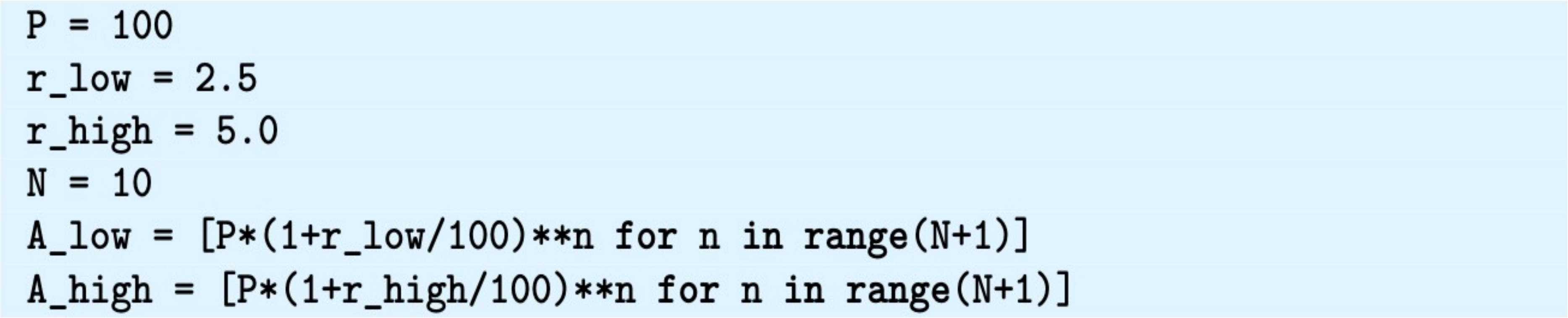
- **Pig Latin**: Given a string containing English words, translate the sentence into Pig Latin. Spilts() will separate a sting into a list of words. Modify each word and then re-assemble into a final String.
  - If word starts with consonant, place starting letter on tail and append ”ay”
  - If word start with vowel, append “vay”
  - If word is less than 3 letters, do not modify.

Advanced List Stuff: List Comprehension



```
P = 100
r_low = 2.5
r_high = 5.0
N = 10
A_high = []
A_low = []
for n in range(N+1):
    A_low.append(P*(1+r_low/100)**n)
    A_high.append(P*(1+r_high/100)**n)
```

Iterating over list and adding new items via append()



```
P = 100
r_low = 2.5
r_high = 5.0
N = 10
A_low = [P*(1+r_low/100)**n for n in range(N+1)]
A_high = [P*(1+r_high/100)**n for n in range(N+1)]
```

Use list comprehension by defining expression that fills list