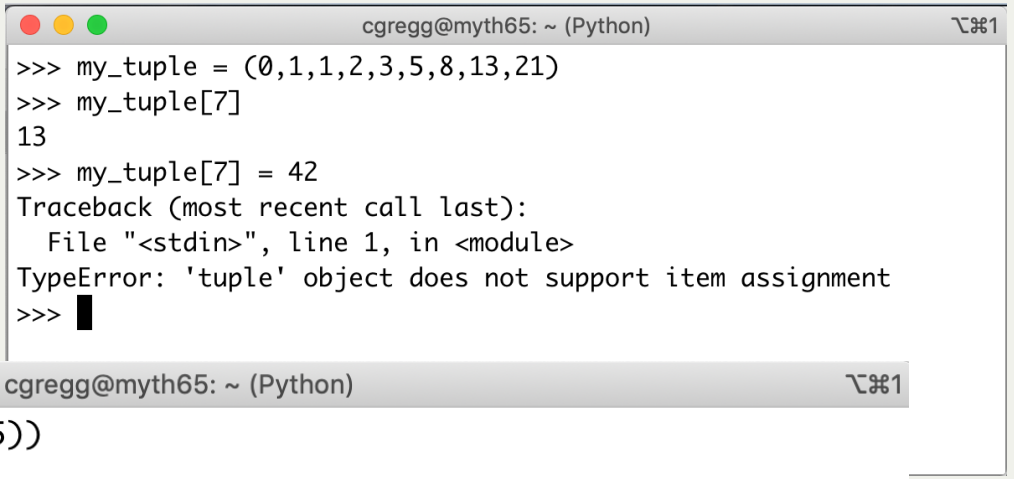
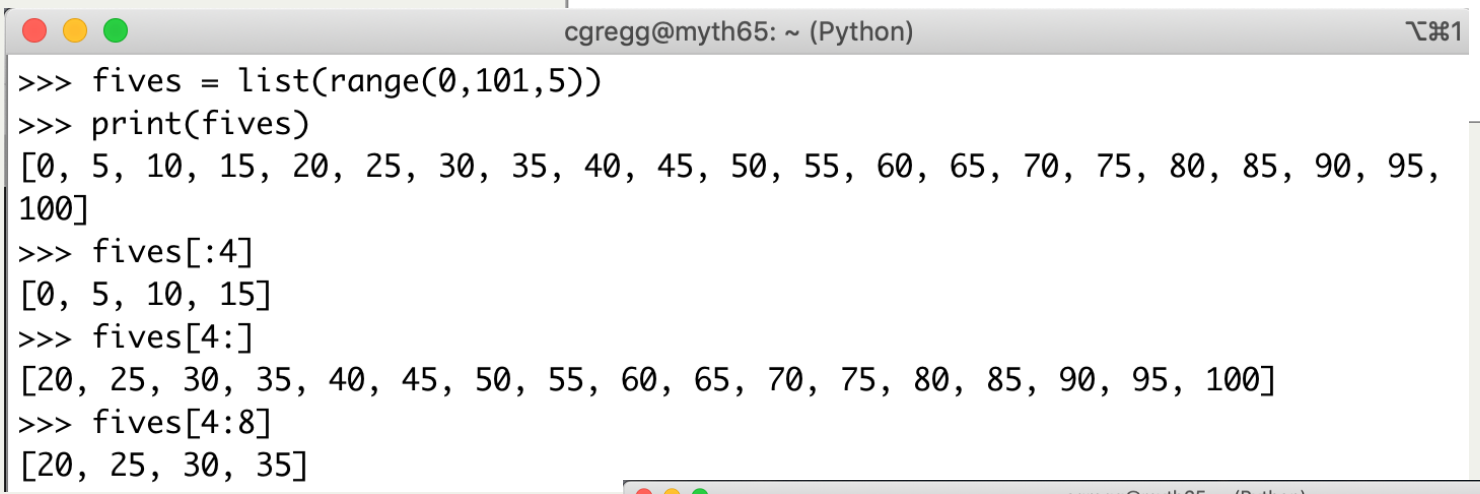
Python Extras: Things you should know

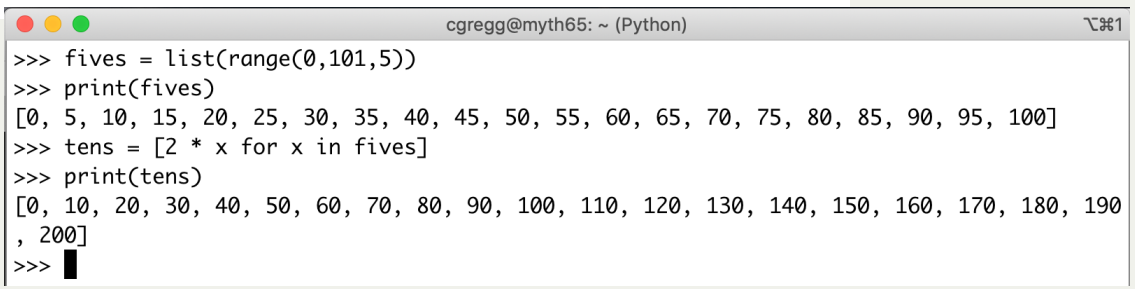
CS 106A

Stanford University

Chris Gregg







PDF of this presentation

Python Extras: Things you should know

- There are a number of Python features that we haven't covered in class, but you should know about them! We're going to go over the following features today, and you will see them quite often when you read Python, and they will be useful when you write Python programs

1. List comprehensions
2. The **`zip`** function
3. Python Sets
4. Dictionary comprehensions
5. The **`enumerate`** function
6. **`try`** / **`except`**

Python: Things you should know: List Comprehensions

One of the slightly more advanced features of Python is the *list comprehension*. List comprehensions act a bit like a for loop, and are used to produce a list in a concise way.

A list comprehension consists of brackets containing an expression followed by a **`for`** clause, then zero or more **`for`** or **`if`** clauses.

The result is a new list resulting from evaluating the expression in the context of the **`for`** and **`if`** clauses which follow it.

The list comprehension always returns a list as its result. Example:

```
>>> my_list
[15, 50, 10, 17, 5, 29, 22, 37, 38, 15]
>>> new_list = [2 * x for x in my_list]
>>> print(new_list)
[30, 100, 20, 34, 10, 58, 44, 74, 76, 30]
```

In this example, the list comprehension produces a new list where each element is twice the original element in the original list. The way this reads is, "multiply 2 by x for every element, x, in my_list"

Python: Things you should know: List Comprehensions

Example 2:

```
>>> my_list
[15, 50, 10, 17, 5, 29, 22, 37, 38, 15]
>>> new_list = [x for x in my_list if x < 30]
>>> print(new_list)
[15, 10, 17, 5, 29, 22, 15]
```

In this example, the list comprehension produces a new list that takes the original element in the original list only if the element is less than 30. The way this reads is, "select x for every element, x, in my_list if x < 30"

Example 3:

```
>>> my_list
[15, 50, 10, 17, 5, 29, 22, 37, 38, 15]
>>> new_list = [-x for x in my_list]
>>> print(new_list)
[-15, -50, -10, -17, -5, -29, -22, -37, -38, -15]
```

In this example, the list comprehension negates all values in the original list. The way this reads is, "return -x for every element, x, in my_list"

Python: Things you should know: List Comprehensions

Let's do the same conversion for Example 2 from before:

```
>>> my_list
[15, 50, 10, 17, 5, 29, 22, 37, 38, 15]
>>> new_list = [x for x in my_list if x < 30]
>>> print(new_list)
[15, 10, 17, 5, 29, 22, 15]
```

The function:

```
>>> def less_than_30(lst):
...     new_list = []
...     for x in lst:
...         if x < 30:
...             new_list.append(x)
...     return new_list
...
>>> less_than_30(my_list)
[15, 10, 17, 5, 29, 22, 15]
```

You can see that the list comprehension is more concise than the function, while producing the same result.

Python: Things you should know: List Comprehensions

```
>>> my_list
[15, 50, 10, 17, 5, 29, 22, 37, 38, 15]
>>> new_list = [-x for x in my_list]
>>> print(new_list)
[-15, -50, -10, -17, -5, -29, -22, -37, -38, -15]
```

We can re-write list comprehensions as functions, to see how they behave in more detail:

```
>>> my_list
[15, 50, 10, 17, 5, 29, 22, 37, 38, 15]
>>> def negate(lst):
...     new_list = []
...     for x in lst:
...         new_list.append(-x)
...     return new_list
...
>>> negate(my_list)
[-15, -50, -10, -17, -5, -29, -22, -37, -38, -15]
```

Python: Things you should know: List Comprehensions

Open up PyCharm and create a new project called ListComprehensions. Create a new python file called "comprehensions.py".

Create the following program, and fill in the details for each comprehension. We have done the first one for you:

```
1  def main():
2      my_list = [37, 39, 0, 43, 8, -15, 23, 0, -5, 30, -10, -34, 30, -5, 28, 9,
3                 18, -1, 31, -12]
4      print(my_list)
5
6      # create a list called "positives" that contains all the positive values
7      # in my_list
8      positives = [x for x in my_list if x > 0]
9      print(positives)
10
11     # create a list called "negatives" that contains all the negative values
12     # in my_list
13     negatives =
14     print(negatives)
15
16     # create a list called "triples" that triples all the values of my_list
17     triples =
18     print(triples)
19
20     # create a list called "odd_negatives" that contains the negative
21     # value of all the odd values of my_list
22     odd_negatives =
23     print(odd_negatives)
24
25 if __name__ == "__main__":
```

Python: Things you should know: The `zip` function

One function which would have made your life (too) easy for assignment 4 is the **`zip`** function, which takes multiple lists and produces one item from each list as a tuple each time the **`next`** function is used on the result of the **`zip`** function. Example:

```
1 def zip_examples():
2     list1 = ['a', 'b', 'c', 'd', 'e']
3     list2 = [10, 20, 30, 40, 50]
4     for v1, v2 in zip(list1, list2):
5         print(v1, v2)
```

Output:

```
a 10
b 20
c 30
d 40
e 50
```

The **`zip`** function works for as many lists as you want.

Python: Things you should know: The `zip` function

Here is the documentation from **`help(zip)`**:

```
class zip(object)
 |  zip(*iterables) --> zip object
 |
 |  Return a zip object whose .__next__() method returns a tuple where
 |  the i-th element comes from the i-th iterable argument.  The .__next__()
 |  method continues until the shortest iterable in the argument sequence
 |  is exhausted and then it raises StopIteration.
 |
```

In other words: if the lists (or any iterable, like a string) is exhausted, the zipping stops -- the shortest list determines how many tuples we get:

```
1 s1 = "a string"
2 s2 = "second string"
3 for v1, v2 in zip(s1, s2):
4     print(v1, v2)
```

Output:

```
a s
  e
s c
t o
r n
i d
n
g s
```

Python: Things you should know: The `zip` function

You can have as many iterables as you want with **`zip`**:

```
1      s1 = "the"
2      s2 = "cat"
3      s3 = "the"
4      s4 = "bat"
5      s5 = "the"
6      s6 = "rat"
7      print(list(zip(s1, s2, s3, s4, s5, s6)))
```

Output:

```
[('t', 'c', 't', 'b', 't', 'r'), ('h', 'a', 'h', 'a', 'h', 'a'), ('e', 't', 'e', 't', 'e', 't')]
```

Python: Things you should know: Python Sets

There is another collection that we have not talked about that is an important one: the **`set`**. A **`set`** is a collection that cannot hold duplicates.

Example:

```
1 def set_examples():
2     my_set = set()
3     for c in 'Mississippi':
4         my_set.add(c)
5     print(my_set)
```

Output:

```
{'M', 'p', 's', 'i'}
```

You can add duplicate elements as many times as you want to a **`set`**, but the **`set`** only keeps one copy. You can use the **`in`** operator to see if an element is in a set (the output of the following is **`True`**.

```
1 def set_examples():
2     my_set = set()
3     for c in 'Mississippi':
4         my_set.add(c)
5     print('s' in my_set)
```

Python: Things you should know: Python Sets

One interesting feature of a **`set`** is that you can find the intersection, union, and difference of **`set`**s, and also whether a set is a subset or superset (or disjoint, etc.):

```
1 first_three = {1, 2, 3}
2 evens = {2, 4, 6}
3 wholes = {0, 1, 2, 3, 4, 5, 6, 7}
4 print(first_three.union(evens))
5 print(first_three.intersection(evens))
6 print(first_three.difference(evens))
7 print(first_three.issubset(wholes))
8 print(wholes.issuperset(evens))
```

Output:

```
{1, 2, 3, 4, 6}
{2}
{1, 3}
True
True
```

(note: none of the functions above change the value of either set)

Python: Things you should know: Dictionary Comprehensions

In the same way that list comprehensions are used to convert one list into another list, dictionary comprehensions can be used to convert one dictionary into another.

Dictionary comprehensions use the **`items`** function of a dictionary, and have the form:

```
dict_variable = {key:value for (key,value) in dictonary.items()}
```

Example:

```
def dictionary_comprehensions():
    dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

    # Double each value in the dictionary
    double_dict1 = {k: v * 2 for (k, v) in dict1.items()}
    print(f"original dict: {dict1}")
    print(f"new dict:      {double_dict1}")
```

Output:

```
original dict: {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
new dict:      {'a': 2, 'b': 4, 'c': 6, 'd': 8, 'e': 10}
```

Python: Things you should know: Dictionary Comprehensions

You can get fancy and modify the keys:

```
1 d = {'a': 5, 'b': 6, 'c': 7}
2 d_xkeys = {k + 'x': v for (k, v) in d.items()}
3 print(d_xkeys)
```

Output:

```
{'ax': 5, 'bx': 6, 'cx': 7}
```

You do have to be careful: remember that dictionaries cannot have duplicate keys:

```
1 d = {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
2 d_mod = {k % 3: v for (k, v) in d.items()}
3 print(f"original dict: {d}")
4 print(f"new dict:      {d_mod}")
```

Output:

```
original dict: {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
new dict:      {1: 'd', 2: 'e', 0: 'c'}
```

Python: Things you should know: The enumerate function

If you want to loop through a collection and you want both the index and the value in the collection, you can do that the way we've seen before:

```
1 def enumerate_examples():
2     xyz = ['x', 'y', 'z']
3     print("old way:")
4     for i in range(len(xyz)):
5         value = xyz[i]
6         print(f"{i}: {value}")
```

Or you can use the **`enumerate`** function, which gets both parts for you:

```
1 def enumerate_examples():
2     xyz = ['x', 'y', 'z']
3     print("using enumerate:")
4     for i, value in enumerate(xyz):
5         print(f"{i}: {value}")
```

Output:

```
old way:
0: x
1: y
2: z

new way:
0: x
1: y
2: z
```

I would say that I use the **`enumerate`** function in about 30% of all the python programs I write.

Python: Things you should know: exceptions

Sometimes, your program does not behave the way you want it to, through no real fault of your own. For example, let's say you try to open a file for reading that doesn't exist:

```
>>> with open("my_missing_file.txt", "r") as f:
...     for line in f:
...         print(line)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'my_missing_file.txt'
```

In this case, your program would crash! We can avoid that crash by *catching the exception*, meaning that we have Python tell us that there has been an error. For example:

```
>>> try:
...     with open("my_missing_file.txt", "r") as f:
...         for line in f:
...             print(line)
... except:
...     print("Something went wrong when trying to open the file!")
...
Something went wrong when trying to open the file!
>>>
```

We control the message, and we can recover from the error, instead of crashing the program.

Python: Things you should know: exceptions

```
>>> try:
...     with open("my_missing_file.txt", "r") as f:
...         for line in f:
...             print(line)
... except:
...     print("Something went wrong when trying to open the file!")
...
Something went wrong when trying to open the file!
>>>
```

When you have a simple **`except`** statement, this will catch *any* error, which is often not what you want to do. You should try to catch the actual exception that you expect. For example:

```
>>> try:
...     with open("my_missing_file.txt", "r") as f:
...         for line in f:
...             print(line)
... except FileNotFoundError:
...     print("Something went wrong when trying to open the file!")
...
Something went wrong when trying to open the file!
```

We knew that there could be a **`FileNotFoundError`**, so we caught it directly (how did we know? We tried it and saw that error, on the last slide!)

Python: Things you should know: exceptions

You can catch any error that the system produces, as long as you know what you are looking for. For example:

```
>>> a = int(input("Please enter an integer: "))
Please enter an integer: October
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'October'
>>> valid_input = False
>>> while not valid_input:
...   try:
...     a = int(input("Please enter an integer: "))
...     valid_input = True
...   except ValueError:
...     print("That wasn't a valid integer...")
...
Please enter an integer: October
That wasn't a valid integer...
Please enter an integer: Bob
That wasn't a valid integer...
Please enter an integer: -3
```

In this case, we continued the program, even though the user kept typing non-integer inputs. We saved ourselves from a crash!