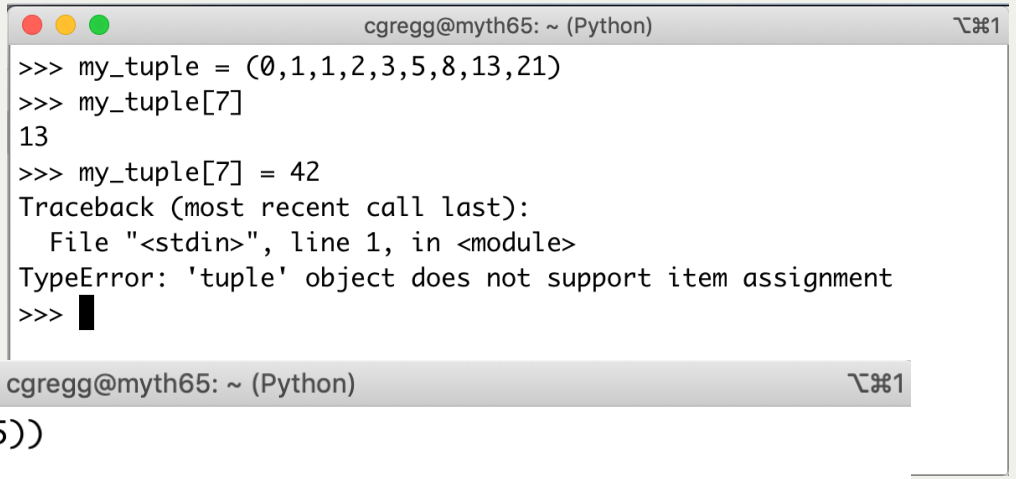
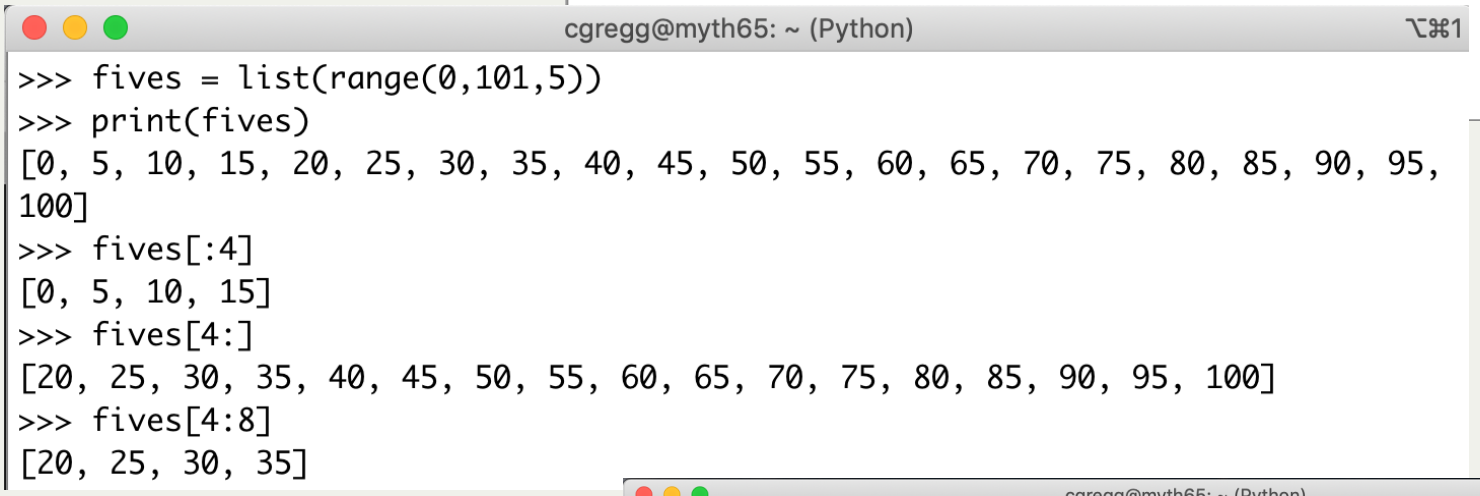
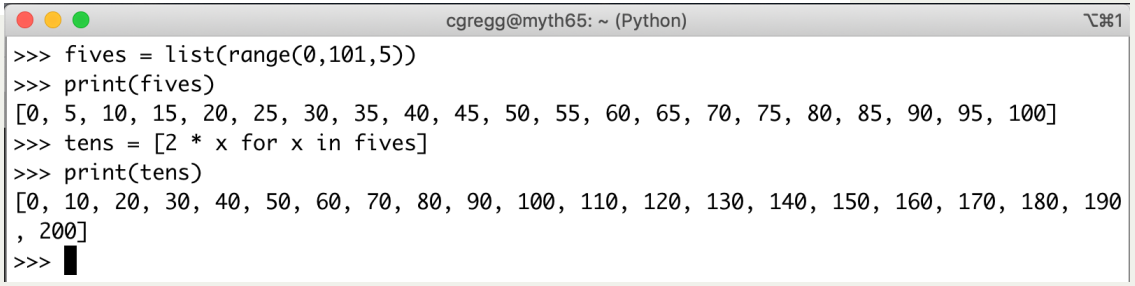
Lecture 4: Tuples, list slicing, list comprehension, strings

CS5001 / CS5003:
Intensive Foundations
of Computer Science







PDF of this presentation

Lecture 4: Announcements

- We are a bit behind on grading! Very sorry about that -- we are trying to work on getting things graded quickly. We will catch up soon!
- Let's chat about how the class is going.
  - Take a few minutes now and write down three things that are going well, and three things that could be better. Don't worry about being frank about it -- I'd like some real feedback (we want to make the course better!)
  - After you write them down, talk with a neighbor about them. If you agree on one or more items, flag them and bring them up.
  - Let's talk about them as a class
  - I will also have an anonymous feedback site set up so you can put comments there that you don't want to discuss in class

Lecture 4: Tuples

In Python, a *tuple* is an *immutable (unchangeable)* collection of elements, much like a list. Once you create a tuple, you cannot change any of the elements.

To create a tuple, you surround your elements with parentheses:

```
>>> animals = ('cat', 'dog', 'aardvark', 'hamster', 'ermine')
>>> print(animals)
('cat', 'dog', 'aardvark', 'hamster', 'ermine')
>>> type(animals)
<class 'tuple'>
>>>
```

Tuples act very much like lists -- you can iterate over them, and you can access the elements with bracket notation:

```
>>> print(animals[3])
hamster
>>> for animal in animals:
...   print(f"Old Macdonald had a/an {animal}!")
...
Old Macdonald had a/an cat!
Old Macdonald had a/an dog!
Old Macdonald had a/an aardvark!
Old Macdonald had a/an hamster!
Old Macdonald had a/an ermine!
>>>
```

Lecture 4: Tuples

If you don't put the parentheses, but have a comma-separated collection of values, they become a tuple, by default:

```
1 >>> 4, 5, 6, 7
2 (4, 5, 6, 7)
3 >>>
```

Because a tuple is a collection of elements separated by parentheses, you can't simply get a single-value tuple with parentheses, because this acts like a regular value in parentheses:

```
 1 >>> singleton = (5)
 2 >>> print(singleton)
 3 5
 4 >>> type(singleton)
 5 <class 'int'>
 6 >>> singleton = (5,)
 7 >>> print(singleton)
 8 (5,)
 9 >>> type(singleton)
10 <class 'tuple'>
11 >>>
```

Instead, you have to put a comma after a single value if you want it to be a single-value tuple, as on line 6. Yes, this looks odd, but that's the way you have to do it.

Notice that when you print out a single-value tuple, it also puts the comma after the single value, to denote that it is a tuple.

Lecture 4: Tuples

Remember when I said that functions can have only a single return value? Well, they can, but that value could be a tuple:

```
 1 >>> def quadratic_equation(a, b, c):
 2 ...   posX = (-b + math.sqrt(b * b - 4 * a * c)) / (2 * a)
 3 ...   negX = (-b - math.sqrt(b * b - 4 * a * c)) / (2 * a)
 4 ...   return posX, negX
 5 ...
 6 >>> import math
 7 >>> quadratic_equation(6, 11, -35)
 8 (1.6666666666666667, -3.5)
 9 >>> x1, x2 = quadratic_equation(5, -2, -9)
10 >>> print(f"The two solutions to the quadratic equation for 5x^2 - 2x -9 are: "
11           f"{x1} and {x2}.")
12 The two solutions to the quadratic equation for 5x^2 - 2x -9 are:
13   1.5564659966250536 and -1.1564659966250537.
14 >>>
```

Notice that you can return a tuple by simply returning two (or more) values separated by commas, as in line 4 above.

You can capture tuple return values separately into variables, as in line 9 above.

Lecture 4: Tuples

As shown on the last slide, you can capture the values of a tuple with a comma separated list of variables:

```
>>> inner_planets = ['Mercury', 'Venus', 'Earth', 'Mars']
>>> mercury, venus, earth, mars = inner_planets
>>> print(mercury)
Mercury
```

This ability allows you to do some interesting things with tuples. What does the following do?

```
>>> x = 5
>>> y = 12
>>> x, y = y, x
```

Lecture 4: Tuples

As shown on the last slide, you can capture the values of a tuple with a comma separated list of variables:

```
>>> inner_planets = ['Mercury', 'Venus', 'Earth', 'Mars']
>>> mercury, venus, earth, mars = inner_planets
>>> print(mercury)
Mercury
```

This ability allows you to do some interesting things with tuples. What does the following do?

```
>>> x = 5
>>> y = 12
>>> x, y = y, x
```

It swaps the values!

```
>>> x = 5
>>> y = 12
>>> x, y = y, x
>>> print(x)
12
>>> print(y)
5
>>>
```

Lecture 4: Tuples

Another use for a tuple is to *gather* arguments for a function that takes a variable number of arguments:

```
>>> def product(*args):
...     prod = 1
...     for n in args:
...         prod *= n
...     return prod
...
>>> product(5,4,3)
60
>>> product(5,4,3,2)
120
```

The opposite of gather is *scatter*. If you have a sequence of values you want to pass to a function that takes multiple arguments, you can do so. We have seen the **`divmod`** function before, which takes two arguments. If you have a tuple you want to use in the **`divmod`** function, you can do so like this:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1

>>> divmod(*t)
(2, 1)
```

Lecture 4: Tuple and List Slicing

One very powerful Python feature is the *slice*, which works for tuples, lists, and strings. A slice is a segment of the collection.

The operator `[n:m]` returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters.

```
>>> lst = [1,4,5,9,12]
>>> tup = [15,8,3,27,18,50,43]
>>> str = "abcdefghijklmnopqrstuvwxyz"
>>>
>>> lst[:3]
[1, 4, 5]
>>> lst[0:3]
[1, 4, 5]
>>> lst[3:]
[9, 12]
>>> tup[6:8]
[43]
>>> tup[4:7]
[18, 50, 43]
>>> str[10:15]
'klmno'
>>> str[11:16]
'lmnop'
>>>
```

- Notice that you can omit either the first or the second number, although you do need to keep the colon.
- You can also have an end value that is after the end of the collection, and Python just stops at the end.

Lecture 4: Tuple and List Slicing

You can also use the optional third argument to a slice: the increment.

This behaves just like the increment in the **`range()`** function:

```
 1 >>> fib = [0,1,1,2,3,5,8,13,21]
 2 >>> fib[3:6]
 3 [2, 3, 5]
 4 >>> fib[3:6:2]
 5 [2, 5]
 6 >>> fib[3:7]
 7 [2, 3, 5, 8]
 8 >>> fib[3:7:2]
 9 [2, 5]
10 >>>
```

- You can also increment *backwards*, as you can do in the **`range()`** function:

```
1 >>> fib[-1:-5:-1]
2 [21, 13, 8, 5]
```

- This means: slice from index -1 (the 21) to one before index -5 (the 5), and go backwards by 1.

- The easiest way to *reverse* an entire list is as follows:

```
1 >>> fib[::-1]
2 [21, 13, 8, 5, 3, 2, 1, 1, 0]
```

Lecture 4: Tuple and List Slicing

Slicing allows you to chop a collection into sections, and then you can put those sections back together in any way you want. For example, the last part of the lab last week discussed "sliding" a string (we will discuss strings in more detail soon). To slide (also called *rotate*), we rotate the values around in the list. For example, `[0,1,1,2,3,5,8,13,21]` rotated by 3 would be `[8,13,21,0,1,1,2,3,5]`.

- We can manually rotate by 3 as follows (you can add two lists together and they *concatenate*)

```
1 >>> fib = [0,1,1,2,3,5,8,13,21]
2 >>> fib[6:] + fib[:6]
3 [8, 13, 21, 0, 1, 1, 2, 3, 5]
```

- We can make this a bit more general, as follows:

```
1 >>> fib[len(fib)-3:] + fib[:len(fib)-3]
2 [8, 13, 21, 0, 1, 1, 2, 3, 5]
```

Lecture 4: Tuple and List Slicing

Let's write a generic function to rotate by any amount, up to the length of the collection. Here is a first attempt:

```
>>> def rotate(lst, rot_amt):
...   return lst[len(lst)-rot_amt:] + lst[:len(lst)-rot_amt]
...
>>> rotate(fib,3)
[8, 13, 21, 0, 1, 1, 2, 3, 5]
>>> rotate(fib,4)
[5, 8, 13, 21, 0, 1, 1, 2, 3]
>>>
```

We can actually make things a bit cleaner:

```
>>> def rotate(lst, rot_amt):
...   return lst[-rot_amt:] + lst[:-rot_amt]
```

If we want to make this work for all values of rot_amt (instead of just the values less than the length of the collection):

```
>>> def rotate(lst, rot_amt):
...   return lst[-rot_amt % len(lst):] + lst[:-rot_amt % len(lst)]
```

Lecture 4: 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"

Lecture 4: 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"

Lecture 4: 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.

Lecture 4: 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]
```

Lecture 4: List Comprehensions: your turn!

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:

```
if __name__ == "__main__":
    my_list = [37, 39, 0, 43, 8, -15, 23, 0, -5, 30, -10, -34, 30, -5, 28, 9,
               18, -1, 31, -12]
    print(my_list)

    # create a list called "positives" that contains all the positive values
    # in my_list
    positives = [x for x in my_list if x > 0]
    print(positives)

    # create a list called "negatives" that contains all the positive values
    # in my_list
    negatives =
    print(negatives)

    # create a list called "triples" that triples all the values of my_list
    triples =
    print(triples)

    # create a list called "odd_negatives" that contains the negative
    # value of all the odd values of my_list
    odd_negatives =
    print(odd_negatives)
```

Lecture 4: Strings

Although we have already discussed strings to some extent in class, let's go into more detail about what a string is, and some of the things you can do with strings in Python.

A string is a *sequence of characters*. Each character has its own ASCII code (as we discussed in lab), which is just a number. We can define strings in Python with either single quotes (') or double-quotes (''), and we can *nest* the quotes. Examples:

```
>>> str = 'This is also a string'
>>> str2 = 'This is also a string'
>>> str3 = 'We can "nest" strings, as well'
>>> print(str3)
We can "nest" strings, as well
```

We can access individual characters in a string with bracket notation, just like with a list. *But*, we cannot change strings -- they are *immutable*:

```
>>> str = "We cannot modify strings"
>>> print(str[3])
c
>>> str[3] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Lecture 4: Strings

If you *do* want to change a letter in a string, you must build a new string. But, because strings allow slicing (like lists), we can do something like this:

```
>>> state = "Calefornia"
>>> state = state[:3] + 'i' + state[4:]
>>> print(state)
California
```

There are other ways to do the same thing, too. For example, we could convert the string into a list, and then modify the character. Then, we can use a method called **`join`** to convert the list back into a string:

```
>>> state = "Calefornia"
>>> temp = list(state)
>>> temp[3] = 'i'
>>> state = ''.join(temp)
>>> print(state)
California
>>>
```

What is this **`join`** function? Let's look:

```
help(''.join)
Help on built-in function join:

join(iterable, /) method of builtins.str instance
    Concatenate any number of strings.

    The string whose method is called is inserted in
        between each given string.
    The result is returned as a new string.

    Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
(END)
```

Lecture 4: Strings

The **`join`** method works on any iterable (e.g., a list, tuple, or string):

```
>>> '.'.join("abcde")
'a.b.c.d.e'
>>> 'BETWEEN'.join(["kl","mn","op"])
'klBETWEENmnBETWEENop'
>>> ' BETWEEN '.join(("kl","mn","op"))
'kl BETWEEN mn BETWEEN op'
>>>
```

It looks like there are methods that strings can use...let's see what other ones there are:

```
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
 'translate', 'upper', 'zfill']
```

Wow! There are a lot of functions! (starting at **`capitalize`**, above)

Lecture 4: Strings

```
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
 'translate', 'upper', 'zfill']
```

We will randomly choose who investigates which functions, and then we are going to investigate them for about ten minutes. Then, we are all going to take turns explaining them to the class.

```
>>> s = "this is a string"
>>> help(s.capitalize)

Help on built-in function capitalize:

capitalize() method of builtins.str instance
    Return a capitalized version of the string.

    More specifically, make the first character have upper case and the rest lower
    case.

>>> print(s.capitalize())
This is a string
>>>
```

Lecture 4: Strings

```
import random

if __name__ == "__main__":
    string_functions = ['capitalize', 'casefold', 'center', 'count', 'encode',
                        'endswith', 'expandtabs', 'find', 'format',
                        'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
                        'isdecimal', 'isdigit', 'isidentifier', 'islower',
                        'isnumeric', 'isprintable', 'isspace', 'istitle',
                        'isupper', 'join', 'ljust', 'lower', 'lstrip',
                        'maketrans', 'partition', 'replace', 'rfind', 'rindex',
                        'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
                        'splitlines', 'startswith', 'strip', 'swapcase',
                        'title', 'translate', 'upper', 'zfill']
    participants = ["Adam", "Christina", "Isaac", "Tiezhou", "Bernard", "James",
                    "Vera", "Lei", "Ely", "Tianhui", "Edmond", "Amelia",
                    "Charlene", "Becky", "Jessica", "Yonnie", "Mac", "Zihao",
                    "Kamilah", "Alex", "Kristina", "Chris"]
    random.shuffle(string_functions)
    random.shuffle(participants)

    investigations = []
    for i, func in enumerate(string_functions):
        investigations.append(
            f"{participants[i % len(participants)]} investigates {func}")

    investigations.sort()
    for investigation in investigations:
        print(investigation)
```

Lecture 4: Strings

Because strings are *iterable*, we can use them in a for loop, which accesses one character at a time:

```
>>> str = "INCEPTION"
>>> for c in str:
...   print(f"{c} ",end='')
...
I N C E P T I O N >>>
```

You can compare two strings with the standard comparison operators. Examples:

```
>>> "zebra" < "Zebra"
False
>>> "Zebra" < "Giraffe"
False
>>> "Giraffe" < "Elephant"
False
>>> "elephant" < "zebra"
True
>>> "elephant" < "elephants"
True
```

- Uppercase letters are *less* than lowercase letters (why? ASCII! `ord('A') == 65` and `ord('a') == 97`).
- The comparison checks one character from each string at a time.
- If two strings are the same until one ends, the shorter string is less than the longer string.