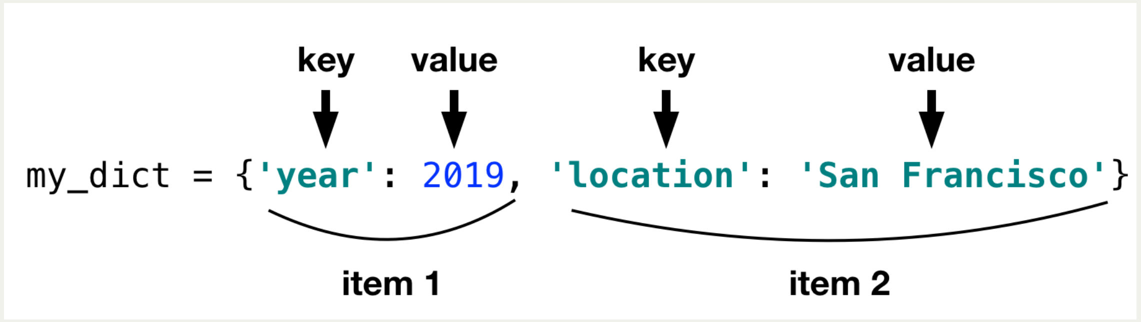
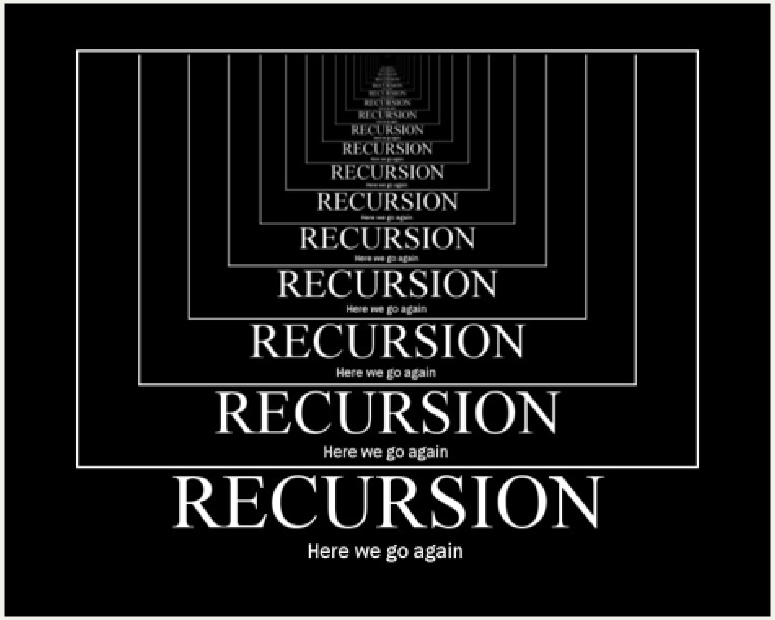
Lecture 5: Dictionaries and Recursion

CS5001 / CS5003:
Intensive Foundations
of Computer Science



```
my_dict = {'year': 2019, 'location': 'San Francisco'}
```



PDF of this presentation

Lecture 5: Practice!

Let's practice a few topics that I think some students are still unclear about:

- Looping a certain number of times
- Printing a comma-separated list without the last comma
- Breaking out of a **`while True`** loop

Practice 1: Write a loop to print a string, **`s`**, X times (once per line), where X is the length of the string. E.g., if the string is "**`bats`**":

```
bats
bats
bats
bats
```

Lecture 5: Practice!

Let's practice a few topics that I think some students are still unclear about:

- Looping a certain number of times
- Printing a comma-separated list without the last comma
- Breaking out of a **`while True`** loop

Practice 1: Write a loop to print a string, **`s`**, X times (once per line), where X is the length of the string. E.g., if the string is "**`bats`**":

```
bats
bats
bats
bats
```

```python
# solution
for i in range(len(s)):
  print(s)
```

Lecture 5: Practice!

Let's practice a few topics that I think some students are still unclear about:

- Looping a certain number of times
- Printing a comma-separated list without the last comma
- Breaking out of a **`while True`** loop

Practice 2: print a list, **`lst`**, one value at a time and comma separated, without the last comma.

Lecture 5: Practice!

Let's practice a few topics that I think some students are still unclear about:

- Looping a certain number of times
- Printing a comma-separated list without the last comma
- Breaking out of a **`while True`** loop

Practice 2: print a list, **`lst`**, one value at a time and comma separated, without the last comma.

```
# solution #1
lst = ['cat', 'dog', 'bat', 'rat']

for i, val in enumerate(lst):
  if i == len(lst) - 1:
    print(val)
  else:
    print(f"{val}, ", end='')
```

```
# solution #3
lst = ['cat', 'dog', 'bat', 'rat']

print(', '.join(lst))
```

```
# solution #2
lst = ['cat', 'dog', 'bat', 'rat']

for v in lst[:-1]:
  print(f"{v}, ", end='')
  print(lst[-1])
```

Lecture 5: Practice!

Let's practice a few topics that I think some students are still unclear about:

- Looping a certain number of times
- Printing a comma-separated list without the last comma
- Breaking out of a **`while True`** loop

Practice 3: Print random numbers between 0 and 99 until two numbers in a row are 90 or above

Lecture 5: Practice!

Let's practice a few topics that I think some students are still unclear about:

- Looping a certain number of times
- Printing a comma-separated list without the last comma
- Breaking out of a **`while True`** loop

Practice 3: Print random numbers between 0 and 99 until two numbers in a row are 90 or above (stop after the two numbers above 90 have been printed)

```
# Solution
last = 0
while True:
  v = random.randint(0,99)
  print(v)
  if v > 90 and last > 90:
    break
  last = v
```

Lecture 5: Calculating the frequencies of letters in a sentence

Challenge: calculate the frequencies of letters in a sentence. For example, the frequencies of letters in "my dog ate my homework" would be:

```
y : 2
  : 4
d : 1
o : 3
g : 1
a : 1
t : 1
e : 2
h : 1
w : 1
r : 1
k : 1
```

There are two y's, four spaces, 1 d, three o's, etc.

How might you write a program to print out those frequencies?

Talk to your neighbor!

Lecture 5: Calculating the frequencies of letters in a sentence

Challenge: calculate the frequencies of letters in a sentence. For example, the frequencies of letters in "my dog ate my homework" would be:

```
y : 2
  : 4
d : 1
o : 3
g : 1
a : 1
t : 1
e : 2
h : 1
w : 1
r : 1
k : 1
```

There are two y's, four spaces, 1 d, three o's, etc.

How might you write a program to print out those frequencies?

Talk to your neighbor!

One solution might look like this:

```python
sentence = "my dog ate my homework"
letter_list = []
letter_freqs = []
for c in sentence:
    if c not in letter_list:
        letter_list.append(c)
        letter_freqs.append(1)
    else:
        letter_freqs[letter_list.index(c)] += 1

for c, freq in zip(letter_list, letter_freqs):
    print(f"{c} : {freq}")
```

This seems kind of messy, though, since we have to keep two simultaneous lists.

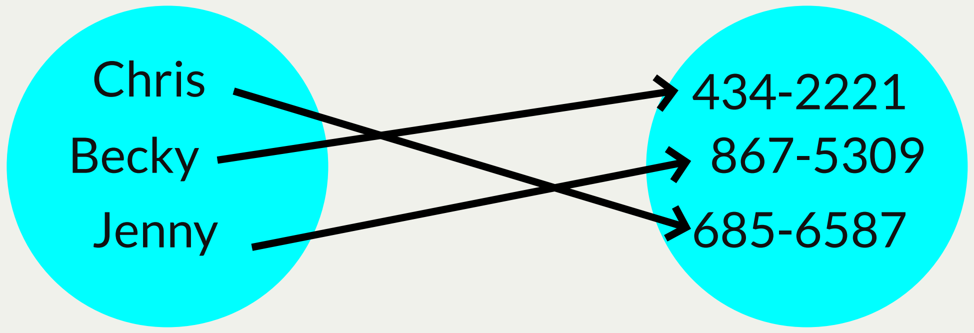
Lecture 5: Dictionaries: a better way!

So far in CS5001, we have talked about the following types of data:

- integers, using **`int`** (e.g., 1, 5, -3, 18)
- floating point numbers, using **`float`** (e.g., **`1.3, 8.2, -14.5`**)
- strings, using either single or double quotes (e.g., **`"hello", "me", 'elephant'`**)
- lists, using square brackets or **`list`** (e.g., **`a = [1, 3, 5, 23, 99]`**)
- tuples, using parentheses (e.g., **`a = (1, 3, 5, 23, 99)`**)

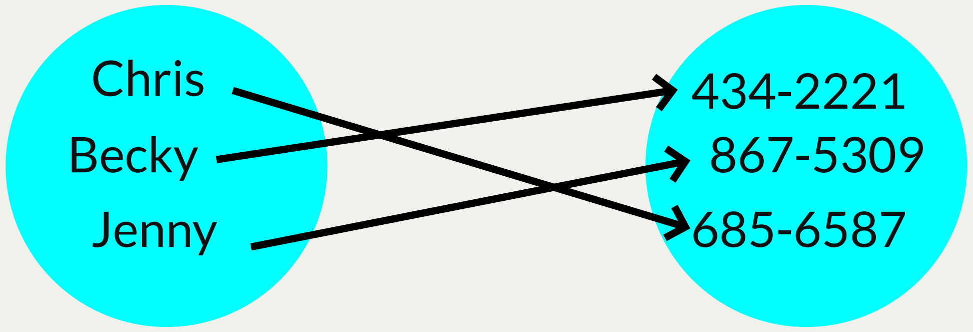
Another very widely used data types is the *dictionary*, or **`dict`**.

A **`dict`** is a set of *key value pairs*. As an example, let's say you wanted to store people's names with their phone numbers. You might have something like this:



In this case, the *keys* are the names, and the *values* are the phone numbers. If you want a value, you ask for it by the key.

Lecture 5: Dictionaries



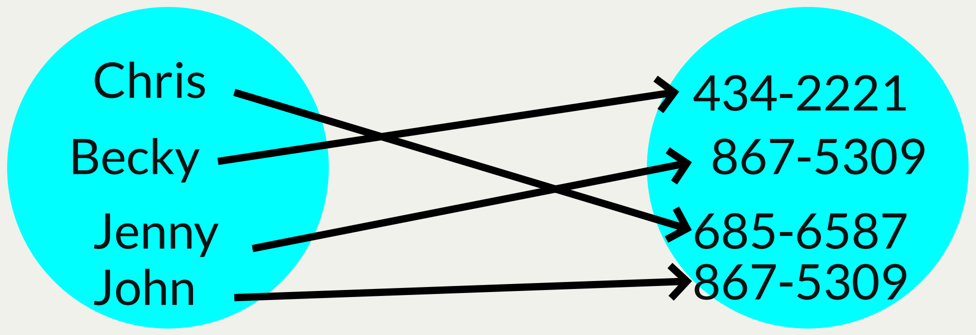
In Python, we can create a dictionary as follows:

```
phonebook = {'Chris' : '685-6587', 'Becky' : '434-2221', 'Jenny' : '867-5309'}
```

To get a value for a key, we use square brackets:

```
>>> phonebook['Chris']
'685-6587'
>>> phonebook['Becky']
'434-2221'
>>> phonebook['Jenny']
'867-5309'
```

Lecture 5: Dictionaries



A dictionary can have the same values, but all keys must be distinct. Let's say we wanted to add "John" to our phonebook:

```
phonebook = {'Chris' : '685-6587', 'Becky' : '434-2221', 'Jenny' : '867-5309'}
phonebook['John'] = '867-5309'
print(phonebook)
{'Chris': '685-6587', 'Jenny': '867-5309', 'John': '867-5309', 'Becky': '434-2221'}
```

Notice that the order when we printed wasn't the order we added the elements in! This is because dictionaries are *not ordered*, and you cannot rely on them to be in a certain order. This is important to remember!

Lecture 5: Dictionaries

Dictionaries are useful for many things. Take a look at this program:

```
sentence = "the quick brown fox jumps over the lazy dog"
frequencies = {}
for c in sentence:
    if c not in frequencies:
        frequencies[c] = 1
    else:
        frequencies[c] += 1
for key, value in frequencies.items():
    print(f"{key} : {value}")
```

What does it do?

Lecture 5: Dictionaries

Dictionaries are useful for many things. Take a look at this program:

```
sentence = "the quick brown fox jumps over the lazy dog"
frequencies = {}
for c in sentence:
    if c not in frequencies:
        frequencies[c] = 1
    else:
        frequencies[c] += 1
for key, value in frequencies.items():
    print(f"{key} : {value}")
```

This program figures out how many of each letter are in the sentence! Output:

```
t : 2
h : 2
e : 3
  : 8
q : 1
u : 2
i : 1
c : 1
k : 1
b : 1
r : 2
o : 4
w : 1
```

```
n : 1
f : 1
x : 1
j : 1
m : 1
p : 1
s : 1
v : 1
l : 1
a : 1
z : 1
y : 1
d : 1
g : 1
```

This is exactly the same program as we wrote earlier, except much less messy. Notice we simply added a new character (as key) to the dictionary if the character wasn't already in the dictionary, and if it was in the dictionary, we simply updated the count.

Lecture 5: Dictionaries

Let's take a look at a larger program, and test a couple of things.

We are going to read in a full book, and count the frequency of the words.

We will have two different functions -- one that uses a list to create the frequency count, and the other that uses a dictionary.

Here is the function using a dictionary:

```python
def get_word_freqs_dict(filename):
    word_frequencies = {}
    word_count = 0
    with open(filename) as f:
        for line in f:
            line = ''.join(ch for ch in line if ch.isalnum() or ch == ' ').lower()
            words = line.split(' ')
            for word in words:
                word_count += 1
                if word_count % 1000 == 0:
                    print(f"Found {word_count} words.")
                if word not in word_frequencies:
                    word_frequencies[word] = 1
                else:
                    word_frequencies[word] += 1
    return word_frequencies
```

Lecture 5: Dictionaries

Here is the function using lists. We convert the result to a dictionary at the end to make the return value of both functions the same:

```python
def get_word_freqs_list(filename):
    word_strings = []
    word_frequencies = []
    word_count = 0
    with open(filename) as f:
        for line in f:
            line = ''.join(ch for ch in line if ch.isalnum() or ch == ' ').lower()
            words = line.split(' ')
            for word in words:
                word_count += 1
                if word_count % 1000 == 0:
                    print(f"Found {word_count} words.")
                if word not in word_strings:
                    word_strings.append(word)
                    word_frequencies.append(1)
                else:
                    word_frequencies[word_strings.index(word)] += 1
    return dict(zip(word_strings, word_frequencies))
```

The functions aren't *that* different, so you might be wondering why we would use one or the other.

Lecture 5: Dictionaries

Here is the driver for the function. Notice that we can easily test both functions by uncommenting / commenting two lines:

```python
if __name__ == "__main__":
    # word_freqs = get_word_freqs_dict(BOOK_FILE)
    word_freqs = get_word_freqs_list(BOOK_FILE)
    print(f"There are {len(word_freqs)} unique words in {TITLE}")
    while True:
        word = input("Word? ").lower()
        if word in word_freqs:
            print(f"'{word}' is found {word_freqs[word]} time(s) in {TITLE}")
```

Let's go see the code and run it...

Lecture 5: Dictionaries

What happened?

The dictionary version was *way* faster!

Why?

- This happened because dictionaries are using a special type of data structure called a *hash map*. You will learn about hash maps in your next course (on data structures), but just know that dictionaries allow fast-lookup of their keys.

In contrast, how would we have to find if a word was in a list?

- We would have to search every item in the list until we found the value. This could take a while for long lists!
- If you want to do this thousands of times for a large data set, you're going to have a bad day!
- Note: you could use a list with a different type of search if you first *sorted* the list, but sorting takes time, too, and keeping a list sorted is also time consuming (when you add another item, etc.)

Lecture 5: Dictionaries

If you want to get a list of just the keys or just the values in a dictionary, you can do it like this:

```
>>> phonebook = {'Chris': '685-6587', 'Jenny': '867-5309', 'John': '867-5309', 'Becky': '434-2221'}
>>> phonebook.keys()
dict_keys(['Chris', 'Jenny', 'John', 'Becky'])
>>> phonebook.values()
dict_values(['685-6587', '867-5309', '867-5309', '434-2221'])
>>>
```

Note: these are actually not lists, but you can treat them as lists. You could also turn them into lists as follows:

```
>>> phonebook = {'Chris': '685-6587', 'Jenny': '867-5309', 'John': '867-5309', 'Becky': '434-2221'}
>>> list(phonebook.keys())
['Chris', 'Jenny', 'John', 'Becky']
>>> list(phonebook.values())
['685-6587', '867-5309', '867-5309', '434-2221']
>>>
```

Remember -- you won't get the keys or values in any particular order (though both lists will be in the same order). Dictionaries are not sorted!

Also notice above that the keys are *always* unique, but the values don't have to be unique (and there are two '867-5309' values in this example).

Lecture 5: Dictionaries

Similar to lists and tuples, dicts can hold different types as either their keys or their values. Frequently, we will want to keep a dict of string keys, with various types for their values. Example:

```python
actor_details = {
    'first_name' : 'Scarlett',
    'last_name' : 'Johansson',
    'movies': ['Lost in Translation', 'Girl with a Pearl Earring',
               'The SpongeBob SquarePants Movie', 'The Avengers'],
    'age' : 34,
    'birthday' : 'November 22, 1984',
}
```

Note that we have nicely formatted this -- one key/value pair per line (except when the lines are too long).

Notice as well that the values are different types -- we have strings, lists, and ints as values.

Finally -- this may seem weird, but we can also end the last item with a comma, even though there aren't more items. This is allowed by Python, and something you will see often (you can also leave off the trailing comma).

Lecture 5: Dictionaries

```python
actor_details = {
    'first_name' : 'Scarlett',
    'last_name' : 'Johansson',
    'movies': ['Lost in Translation', 'Girl with a Pearl Earring',
               'The SpongeBob SquarePants Movie', 'The Avengers'],
    'age' : 34,
    'birthday' : 'November 22, 1984',
}
```

How would we access each item in this dict? With bracket notation:

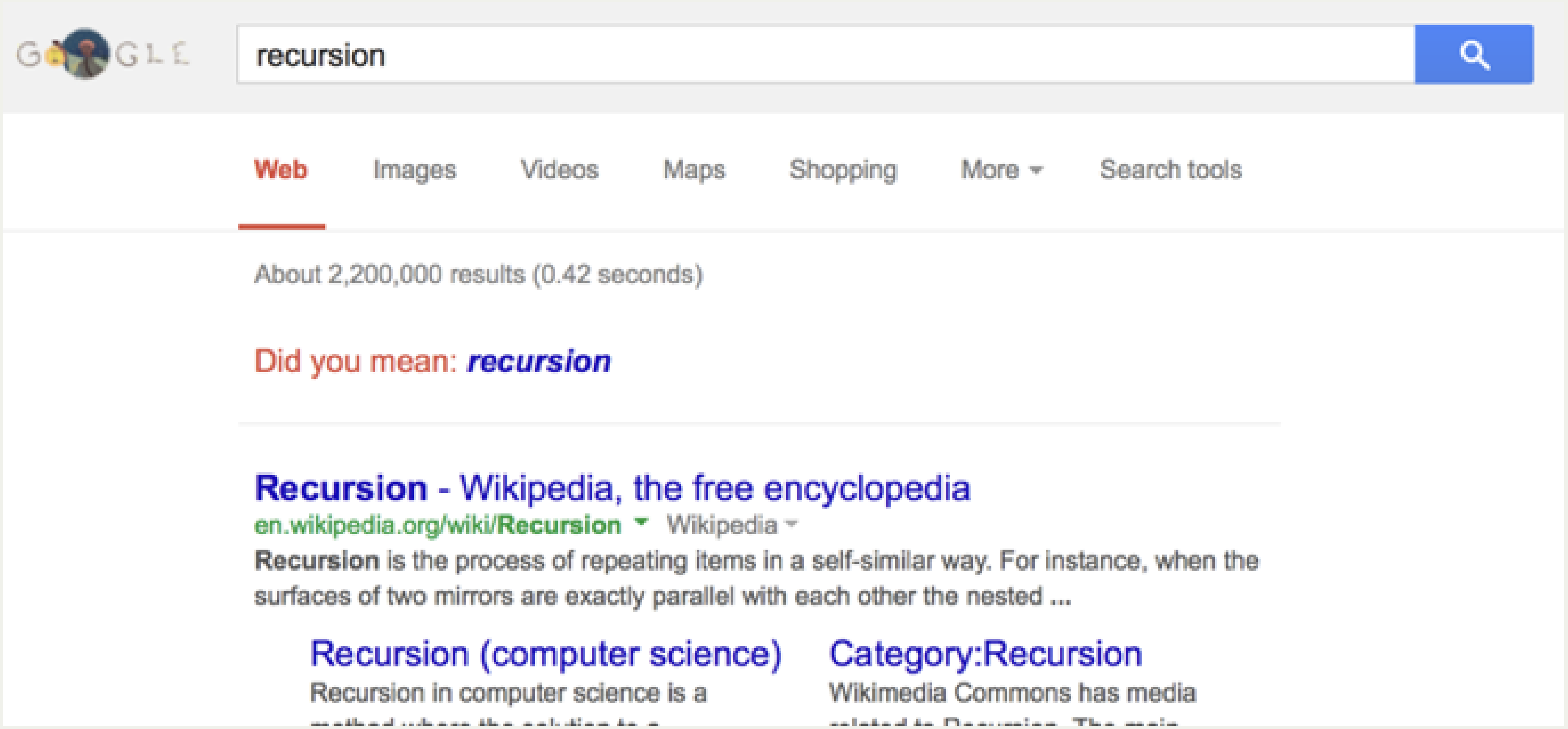
```python
print(f"The actor's name is {actor_details['first_name']} {actor_details['last_name']}, and")
print(f"the actor was in the following films:")
for film in actor_details['movies']:
    print(f"  {film}")
print(f"The actor is {actor_details['age']} years old and was born on {actor_details['birthday']}")
```

Output:

```
The actor's name is Scarlett Johansson, and
the actor was in the following films:
  Lost in Translation
  Girl with a Pearl Earring
  The SpongeBob SquarePants Movie
  The Avengers
The actor is 34 years old and was born on November 22, 1984
```

Lecture 5: Recursion

To understand recursion, you must understand recursion.



recursion

About 2,200,000 results (0.42 seconds)

Did you mean: ***recursion***

Recursion - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/**Recursion** Wikipedia
Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other the nested ...

Recursion (computer science)
Recursion in computer science is a

Category:Recursion
Wikimedia Commons has media

Lecture 5: Recursion

In computer science, *recursion* simply means that a function will *call itself*.

```python
def calc_factorial(n):
    if n == 1:
        return n
    else:
        return n * calc_factorial(n-1)
```

```
>>> print(factorial(5))
120
```

The following program is pretty simple (and will crash!), but it is another example of recursion:

```python
def recurse(x):
  print(x)
  recurse(x + 1)
```

Output:

```
>>> recurse(1)
1
2
3
4
5
...
994
995
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in recurse
  File "<stdin>", line 3, in recurse
  File "<stdin>", line 3, in recurse
  [Previous line repeated 992 more times]
  File "<stdin>", line 2, in recurse
RecursionError: maximum recursion depth exceeded while
calling a Python object 996
```

Lecture 5: Recursion

In computer science, *recursion* simply means that a function will *call itself*.

```python
def calc_factorial(n):
    if n == 1:
        return n
    else:
        return n * calc_factorial(n-1)
```

```
>>> print(factorial(5))
120
```

The following program is pretty simple (and will crash!), but it is another example of recursion:

```python
def recurse(x):
  print(x)
  recurse(x + 1)
```

The function calls itself!

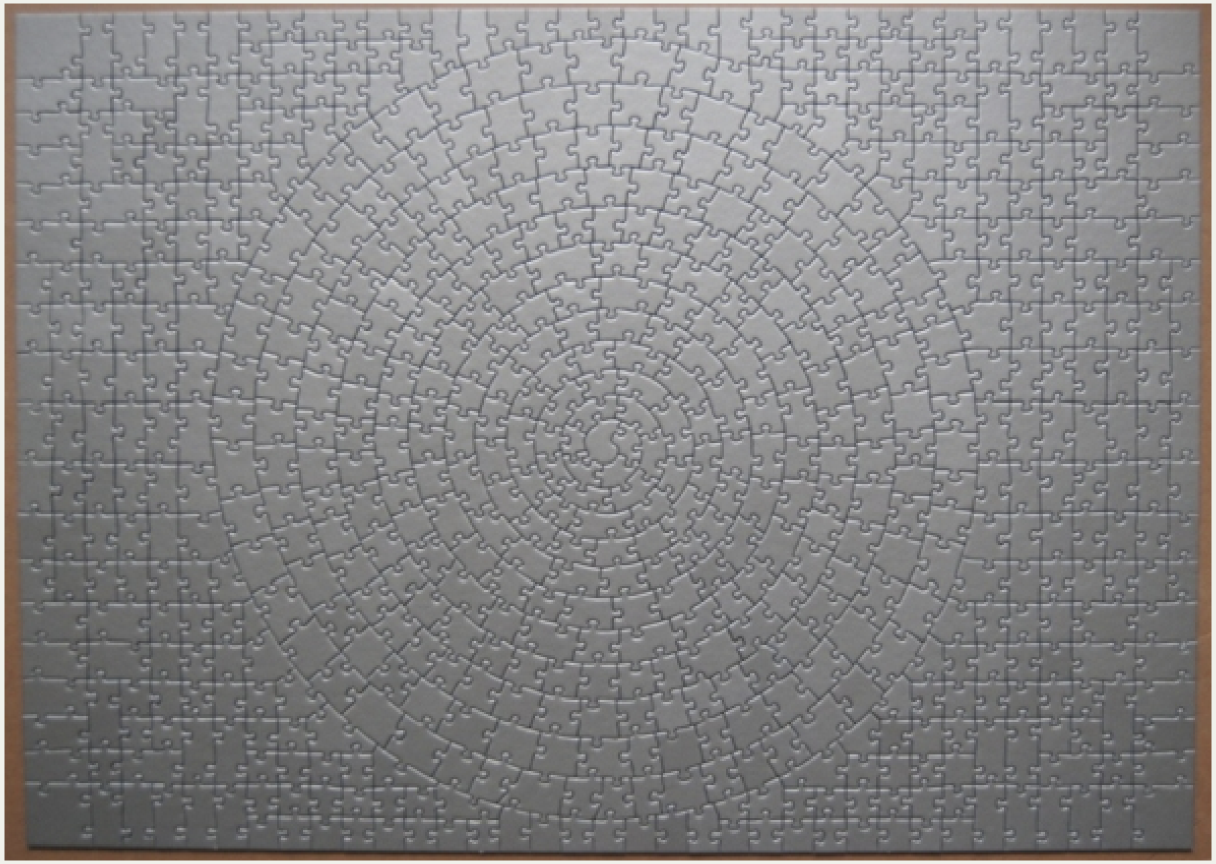
Output:

```
>>> recurse(1)
1
2
3
4
5
...
994
995
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in recurse
  File "<stdin>", line 3, in recurse
  File "<stdin>", line 3, in recurse
  [Previous line repeated 992 more times]
  File "<stdin>", line 2, in recurse
RecursionError: maximum recursion depth exceeded while
calling a Python object 996
```

Lecture 5: Recursion: How to solve a jigsaw puzzle recursively

How to solve a jigsaw puzzle recursively ("solve the puzzle")

- Is the puzzle finished? If so, stop.
- Find a correct puzzle piece and place it.
- Solve the puzzle

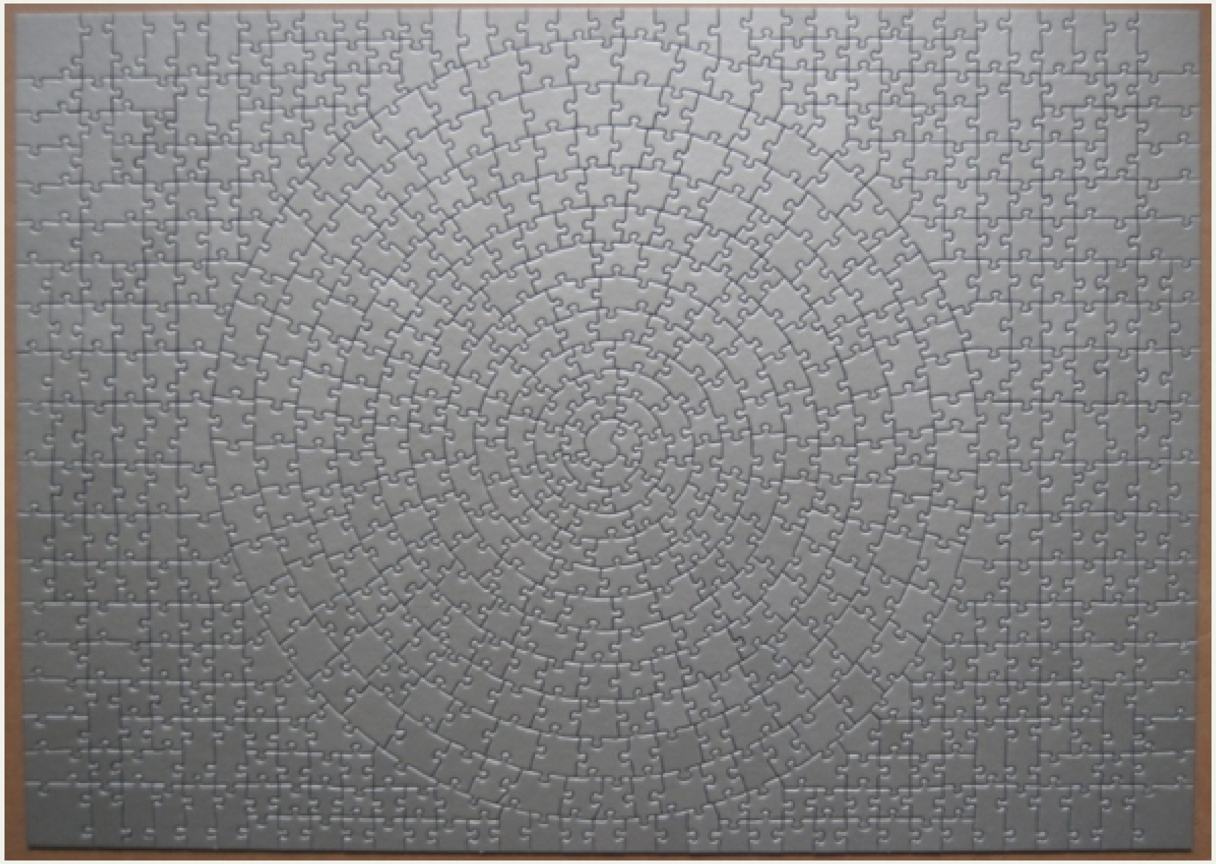


Lecture 5: Recursion: How to solve a jigsaw puzzle recursively

How to solve a jigsaw puzzle recursively ("solve the puzzle")

- Is the puzzle finished? If so, stop.
- Find a correct puzzle piece and place it.
- Solve the puzzle

Ridiculously hard puzzle



Lecture 5: Recursion: The algorithm

In general, to perform a recursive algorithm:

- Check for the *base case*. This is the case where the problem has been solved, and we can stop. We normally we return a result, but not always.
- Work towards the base case.
- Perform a recursive call on our own function.

Lecture 5: Recursion: Counting down

How to count down recursively from N to 0:

- Is N less than 0? If yes, then stop (this the base case)
- Say N (this is working towards the base case, along with the first part of step 3 below)
- Count down from N-1 (this is the recursion!)

```python
1 def count_down(count):
2     if count < 0:
3         return
4     print(count)
5     count_down(count - 1)
```

On line 2, we check the base case: have we reached 0 yet?

On line 3 we print (say) the count

On line 4 we work toward the base case, by calling count_down with one fewer value.

Lecture 5: Recursion: Counting up

What about counting up? Maybe add another variable?

```python
def count_up(count, maximum):
    if count > maximum:
        return
    print(count)
    count_up(count + 1, max)
```

This works, but it is kind of ugly. We have to add a second number. It does have more functionality, though, because we can start the the count at any number.

Lecture 5: Recursion: Counting up

This version is actually very interesting!

```python
def count_up(count):
    if count < 0:
        return
    count_up(count - 1)
    print(count)
```

This works, and is pretty cool. All of the counting happens before anything gets printed. Let's walk through this.

Lecture 5: Recursion: Counting up

Let's look at some other common recursive functions (not all are efficient!!):

- b^n **`power(b,n)`**
- **`factorial(int n);`**
- **`fibonacci(int n);`** (one way is not efficient!)
- **`isPalindrome(string s)`**

Lecture 5: Recursion: Counting up

Let's look at some other common recursive functions (not all are efficient!!):

- b^n `power(b,n)`

```python
def power(b, n):
    if n <= 0:
        return 1
    return b * power(b, n - 1)
```

```
>>> print(power(2,3))
8
```

Lecture 5: Recursion: Counting up

Let's look at some other common recursive functions (not all are efficient!!):

- `factorial(n)`

Recursive:

```python
def calc_factorial(n):
    if n == 1:
        return n
    else:
        return n * calc_factorial(n-1)
```

Iterative:

```python
def calc_factorial_iterative(n):
    product = 1
    while n > 0:
        product *= n
        n -= 1
    return product
```

The iterative function works just fine! Some might argue that the recursive function is more beautiful, but it is not quite as efficient, nor can it calculate factorials that are too big.

Lecture 5: Recursion: Counting up

Let’s look at some other common recursive functions (not all are efficient!!):

- `fibonacci(n)`

Not efficient!

```python
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Efficient with *helper function*

```python
def fibHelper(n, p0, p1):
    if n==1:
        return p1
    return fibHelper(n - 1, p1, p0 + p1)

def fibonacci2(n):
    if n == 0:
        return 0
    return fibHelper(n, 0, 1)
```

Lecture 5: Recursion: Counting up

Let's look at some other common recursive functions (not all are efficient!!):

- `is_palendrome(s)`

```python
def is_palendrome(s):
    # check base case
    if len(s) < 2:
        return True
    # check if the first and second character match
    # if they do, recursively check the rest of the string
    if s[0] == s[-1]:
        return is_palendrome(s[1:-1])
    return False
```