Lesson 2: Iteration, Strings, Lists, and Dictionaries

Professor Jeffrey Regier

adapted from slides by Keith Levin

Recursion

A function is a allowed to call itself, in what is termed **recursion**

```python
def countdown(n):
    if n <= 0:
        print('We have lift off!')
    else:
        print(n)
        countdown(n-1)
```

Countdown calls itself!

But the key is that each time it calls itself, it is passing an argument with its value decreased by 1, so eventually, `n <= 0` is true.

```python
countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

With a small change, we can make it so that `countdown(1)` encounters an **infinite recursion**, in which it repeatedly calls itself.

```python
def countdown(n):
    if n <= 0:
        print('We have lift off!')
    else:
        print(n)
        countdown(n)
```

```python
1 countdown(10)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-163-a972007fb272> in <module>()
----> 1 countdown(10)

<ipython-input-162-33965ef63097> in countdown(n)
      4     else:
      5         print n
----> 6         countdown(n)

... last 1 frames repeated, from the frame below ...

<ipython-input-162-33965ef63097> in countdown(n)
      4     else:
      5         print n
----> 6         countdown(n)

RuntimeError: maximum recursion depth exceeded
```

Repeated actions: Iteration

Recursion is the first tool we've seen for performing repeated operations

But there are better tools for the job: `while` and `for` loops.

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('We have lift off!')
```

```
1 countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

Repeated actions: Iteration

Recursion is the first tool we've seen for performing repeated operations

But there are better tools for the job: `while` and `for` loops.

```python
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('We have lift off!')
```

This block specifies a while-loop. So long as the condition is true, Python will run the code in the body of the loop, checking the condition again at the end of each time through.

Repeated actions: Iteration

Recursion is the first tool we've seen for performing repeated operations

But there are better tools for the job: `while` and `for` loops.

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('We have lift off!')
```

```
countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
We have lift off!
```

Warning: Once again, there is a danger of creating an **infinite loop**. If, for example, `n` never gets updated, then when we call `countdown(10)` , the condition `n>0` will always evaluate to `True`, and we will never exit the while-loop.

Repeated actions: Iteration

```
def collatz(n):
    while n!=1:
        print(n)
        if n % 2 == 0:
            n = n/2
        else:
            n = 3*n+1
```

```
collatz(20)
```

```
20
10
5
16
8
4
2
```

One always wants to try and ensure that a while loop will (eventually) terminate, but it’s not always so easy to know!
https://en.wikipedia.org/wiki/Collatz_conjecture

“Mathematics may not be ready for such problems."
Paul Erdős

Repeated actions: Iteration

We can also terminate a while-loop using the `break` keyword

```
a = 4
x = 3.5
epsilon = 10**-6
while True:
    print(x)
    y = (x + a/x)/2
    if abs(x-y) < epsilon:
        break
    x=y # update to our new estimate
```

```
3.5
2.32142857143
2.02225274725
2.00012243394
2.00000000375
```

The `break` keyword terminates the current loop when it is called.

Newton-Raphson method:
https://en.wikipedia.org/wiki/Newton's_method

Repeated actions: Iteration

We can also terminate a while-loop using the `break` keyword

```
a = 4
x = 3.5
epsilon = 10**-6
while True:
    print(x)
    y = (x + a/x)/2
    if abs(x-y) < epsilon:
        break
    x = y # update to our new estimate
```

```
3.5
2.32142857143
2.02225274725
2.00012243394
2.00000000375
```

Notice that we're not testing for equality here. That's because testing for equality between pairs of floats is dangerous. When I write `x=1/3`, for example, the value of `x` is actually only an approximation to the number 1/3.

Newton-Raphson method: https://en.wikipedia.org/wiki/Newton's_method

Strings in Python

Strings are sequences of characters

Python sequences are 0-indexed. The index counts the offset from the beginning of the sequence. So the first letter is the 0-th character of the string.

Note: in some languages, there's a difference between a character and a string of length 1. That is, the character 'g' and the string "g" are different data types. In Python, no such difference exists. A character is just a one-character string.

```python
animal = 'goat'
letter = animal[1]
letter
```

'o'

```python
animal[0]
```

'g'

```python
animal[1]
```

'o'

```python
animal[2]
```

'a'

```python
animal[3]
```

't'

Strings in Python

Strings are **sequences** of characters

All Python sequences include a **length** attribute, which is the number of elements in the sequence.

```
1 len(animal)
```

```
4
```

If we try to access an element of the sequence that doesn't exist, we get an error.

```
1 animal[4]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-8-71de68f745e5> in <module>()
----> 1 animal[4]

IndexError: string index out of range
```

We can also index into a sequence counting from the end.

```
1 animal[-1]
```

```
't'
```

Strings in Python

```
In [3]: i = 0
        while i < len(animal):
            print(animal[i])
            i = i + 1

g
o
a
t
```

We can index into a sequence using an index variable.

...but there's a better way to perform this operation...

Iterations and traversals: for-loops

```
In [10]: for c in animal:
             print(c)

g
o
a
t
```

```
In [3]: i = 0
        while i < len(animal):
            print(animal[i])
            i = i + 1

g
o
a
t
```

For-loop provides a more concise way to express the pattern on the right.

Selecting subsequences: slices

A segment of a Python sequence is called a **slice**

```python
s = "And now for something completely different"
s[0:7]
```

```
'And now'
```

```python
s[12:21]
```

```
'something'
```

`string[m:n]` picks out the `m`-th character to the `n`-th character, including the `m`-th character, but **not** including the `n`-th character.

Selecting subsequences: slices

`string[:]` picks out the entire string.

`string[x:x]` picks out the `x`-th through `x`-th letters, not including the `x`-th, so this gets the **empty string.**

```
s = "And now for something completely different"
s[:]
```

```
'And now for something completely different'
```

```
s[2:2]
```

```
''
```

Selecting subsequences: slices

A segment of a Python sequence is called a **slice**

```
s = "And now for something completely different"
s[:7]
```

```
'And now'
```

```
s[22:]
```

```
'completely different'
```

```
s[-20:-10]
```

```
'completely'
```

```
s[-20:]
```

```
'completely different'
```

`string[:m]` picks out the subsequence starting at 0 through the `(m-1)`-th character.

`string[m:]` picks out the subsequence starting at the `m`-th character through the end of the sequence.

Slices also work with negative indexing.

Selecting subsequences: slices

`string[:]` picks out the entire string.

```
s = "And now for something completely different"
s[:]
```

```
'And now for something completely different'
```

`string[x:x]` picks out the `x`-th through `x`-th letters, not including the `x`-th, so this gets the **empty string.**

```
s[2:2]
```

```
''
```

The empty string is a string just like any other, but it contains **no letters** and has length 0.

Important concept: immutability

What if I want to change a letter in my string?

Try and assign a different string to a subsequence of a string.

```
1 mystr = 'goat'
2 mystr[0] = 'b'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-cf531ebc1ce4> in <module>()
      1 mystr = 'goat'
----> 2 mystr[0] = 'b'

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

We get an error because strings are **immutable**.
We can't change the value of an *existing* string.

Important concept: immutability

What if I want to change a letter in my string?

```
mystr = 'goat'
mystr = 'b'+mystr[1:]
mystr
```

```
'boat'
```

This avoids the error we saw before because it changes the value of the variable `mystr`, rather than trying to change the contents of a string.

Example: string traversal

```
def count(word, letter):
    cnt = 0
    for c in word:
        if c==letter:
            cnt = cnt+1
    return cnt
```

```
count('banana', 'a')
```

3

```
count('banana', 'z')
```

0

The function `count` makes use of a common pattern, often called a **traversal**. We examine each element of a sequence (i.e., a string), taking some action for each element.

The variable `cnt` keeps a tally of how many times we have seen letter in the string word, so far. We call such a variable a **counter** or an **accumulator**.

Python string methods

Python strings provide a number of built-in operations, called **methods**

```
1 mystr = 'goat'
2 mystr.upper()
```

'GOAT'

```
1 'aBcDeFg'.lower()
```

'abcdefg'

```
1 'banana'.find('na')
```

2

```
1 'goat'.startswith('go')
```

True

`str.upper()` makes all letters in `str` upper case. `str.lower()` is analogous.

`str.find(sub)` finds the index of the first location of the string sub in str.

`str.startswith(sub)` returns `True` if and only if `str` starts with `sub`.

Python string methods

Python strings provide a number of built-in operations, called **methods**

```
mystr = 'goat'
mystr.upper()
```

'GOAT'

```
'aBcDeFg'.lower()
```

'abcdefg'

```
'banana'.find('na')
```

2

```
'goat'.startswith('go')
```

True

This `variable.method()` notation is called **dot notation**, and it is ubiquitous in Python (and many other languages).

A **method** is like a function, but it is provided by an **object**. We'll learn much more about this later in the semester, but for now, it suffices to know that some data types provide what *look* like functions (they take arguments and return values), and we call these function-like things **methods**.

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

Optional arguments: `str.find()`

```
1 'banana'.find('na')
```

2

```
1 'banana'.find('na', 3)
```

4

```
1 'banana'.find('na', 3, 4)
```

-1

```
1 'banana'.find('na', 3, 6)
```

4

The `str.find()` method takes **optional arguments**, which specify where in the string to start looking for a match, and the last index to consider for a match.

Find first occurrence of `'na'`, starting from index `3`.

Find first occurrence of `'na'`, starting from index `3`, and nowhere past `4`.

The documentation writes this method as `str.find(`*`sub`*`[, `*`start`*`[, `*`end`*`]])`. Square brackets indicate optional arguments. In this case, brackets also indicate that with two arguments, the second one will be interpreted as the `start` argument.
https://docs.python.org/3/library/stdtypes.html#string-methods

Searching sequences: the `in` keyword

```
1 'a' in 'banana'
```

True

```
1 'z' in 'banana'
```

False

```
1 'ban' in 'banana'
```

True

```
1 'anan' in 'banana'
```

True

```
1 'zoo' in 'banana'
```

False

`x in y` returns `True` if `x` occurs as a substring of `y`, and `False` otherwise.

Importantly, we can check for a whole substring, making this very similar to `str.find()`.

The **`in`** keyword applies more generally to check whether an object is contained in a sequence. We'll see more examples of this in the future, but for now, we only need to worry about strings.

String Comparison

Sometimes we want to check if two strings are equal

```
'cat' == 'cat'
```

```
True
```

```
'cat' == 'dog'
```

```
False
```

```
'dog' == 'doge'
```

```
False
```

Use the equality operator (==), just like for comparing numbers.

Strings have to match exactly. Substring is not enough!

String Comparison

Sometimes we want to check if two strings are equal

```
1 'cat' == 'cat'
```

True

```
1 'cat' == 'dog'
```

False

```
1 'dog' == 'doge'
```

False

Use the equality operator (==), just like for comparing numbers.

Strings have to match exactly. Substring is not enough!

If we can compare strings with equality, we should be able to compare them with inequalities, too...

String Comparison

We can also compare words under alphabetical ordering

```
'cat' < 'dog'
```

True

```
'cat' >= 'dog'
```

False

```
'dog' < 'doge'
```

True

```
'' < 'goat'
```

True

```
'1' < 'a'
```

True

Words earlier in the dictionary are “smaller” than words later in the dictionary.

The empty string '' comes first in the ordering.

Strings including numbers, symbols, etc. are also ordered.

String Comparison

Important: upper case and lower case letters ordered differently!

```
1 'Cat' == 'cat'
```

```
False
```

```
1 'cat' > 'Cat'
```

```
True
```

Upper case letters are ordered before lower case letters.

For more information:
https://docs.python.org/3/library/stdtypes.html#comparisons

For **much** more information:
https://docs.python.org/3/library/operator.html?highlight=equallity

Python Lists

Strings in Python are “sequences of characters”

But what if I want a sequence of something else?

A vector would be naturally represented as a sequence of numbers

A class roster might be represented as a sequence of strings

Python lists are sequences whose values can be of any data type

We call these list entries the **elements** of the list

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

This is a list of four strings.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

This is a list of nine integers

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

The elements of a list need not be of the same type. Here is a list with a string, an integer and a float.

Constructing Lists

We create a list by putting its elements between square brackets, separated by commas.

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fibonacci = [0, 1, 1, 2, 3, 5, 8, 13, 21]
mixed = ['one', 2, 3.0]
pythagoras = [[3,4,5], [5, 12, 13], [8, 15, 17]]
```

A list can even contain more lists! This is a list of three lists, each of which is a list of three integers.

Constructing Lists

It is possible to construct a list with no elements, the empty list.

```
x = []
x
```

```
[]
```

```
x = list()
x
```

```
[]
```

Two equivalent ways of creating an empty list.

Create a list using square brackets notation, but supply no elements. So `x` is empty.

Use the reserved keyword `list`, which casts to a list. Given no arguments, it creates an empty list.

Accessing List Elements

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
fruits[0]
```

'apple'

```
fruits[1]
```

'orange'

```
fruits[2]
```

'banana'

```
fruits[-1]
```

'kiwi'

We can access individual elements of a list just like a string. This is because both strings and lists are examples of Python **sequences.**

Indexing from the end of the list, just like with strings.

Accessing List Elements

```
1 fruits = ['apple', 'orange', 'banana', 'kiwi']
2 fruits
```

```
['apple', 'orange', 'banana', 'kiwi']
```

Unlike strings, lists are **mutable**. We can change individual elements after creating the list.

```
1 fruits[-1] = 'mango'
2 fruits
```

```
['apple', 'orange', 'banana', 'mango']
```

```
1 mystring = 'goat'
2 mystring[0]='b'
```

Reminder of what happens if we try to do this with a string. This error is because string are **immutable**. Once they're created, they can't be altered.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-b526da741b9a> in <module>()
      1 mystring = 'goat'
----> 2 mystring[0]='b'

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

Lists are sequences, so they have a length

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
len(fruits)
```

4

```
len([])
```

The empty list has length 0, just like the empty string.

0

```
pythagoras = [[3, 4, 5], [5, 12, 13], [8, 15, 17]]
len(pythagoras)
```

One might be tempted to say that `pythagoras` should have length 9, but each element of a list counts only once, even if it is itself a more complicated object!

3

Lists are sequences, so they support the `in` operator

```
1 fruits = ['apple', 'orange', 'banana', 'kiwi']
2 'apple' in fruits
```

True

Just like with strings, `x in y` returns `True` if and only if `x` is an element of `y`.

```
1 'grape' in fruits
```

False

```
1 ['apple', 'orange'] in fruits
```

False

Warning: This contrasts with the string case. Recall that `'ap' in 'apple'` evaluates to `True`. By analogy, this line of code should also evaluate to `True`, but it doesn't, because for lists, the `in` operator only checks elements, not subsequences.

```
1 ['cat','dog'] in [['cat','dog'], ['bird','goat']]
```

True

Common pattern: list traversal

For each element of a list, do something with that element

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
for f in fruits:
    print(f)
```

```
apple
orange
banana
kiwi
```

```
numbers = range(5)
for n in numbers:
    print(2**n)
```

`range(x)` produces a list of the integers 0 to `x-1`.
For more information:
https://docs.python.org/3/library/stdtypes.html#ranges

```
1
2
4
8
16
```

Common pattern: list traversal

For each element of a list, do something with that element

```
fruits = ['apple', 'orange', 'banana', 'kiwi']
for i in range(len(fruits)):
    fruits[i] = fruits[i].upper()

for f in fruits:
    print(f)
```

```
APPLE
ORANGE
BANANA
KIWI
```

Sometimes, we need to be able to index into the list itself, in which case we use a slightly different traversal pattern, in which we iterate an **index variable**, `i` in this example.

Common pattern: list traversal

For each element of a list, do something with that element

```python
fruits = ['apple', 'orange', 'banana', 'kiwi']
for i in range(len(fruits)):
    fruits[i] = fruits[i].upper()

for f in fruits:
    print(f)
```

```
APPLE
ORANGE
BANANA
KIWI
```

Sometimes, we need to be able to index into the list itself, in which case we use a slightly different traversal pattern, in which we iterate an **index variable**, `i` in this example.

Note: this operation is possible because lists are mutable!

List operations: concatenation

List concatenation is similar to strings.

```
1 fibonacci = [0,1,1,2,3,5,8]
2 primes = [2,3,5,7,11,13]
3 fibonacci + primes
```

```
[0, 1, 1, 2, 3, 5, 8, 2, 3, 5, 7, 11, 13]
```

```
1 3*['cat','dog']
```

```
['cat', 'dog', 'cat', 'dog', 'cat', 'dog']
```

These operations are precisely analogous to the corresponding string operations. This makes sense, since both strings and lists are **sequences**.
https://docs.python.org/3/library/stdtypes.html#typesseq

List operations: slices

Also like strings, it is possible to select **slices** of a list

```
animals = ['cat', 'dog', 'goat', 'bird', 'llama']
animals[1:3]
```

```
['dog', 'goat']
```

```
animals[3:]
```

```
['bird', 'llama']
```

```
animals[:2]
```

```
['cat', 'dog']
```

```
animals[:]
```

```
['cat', 'dog', 'goat', 'bird', 'llama']
```

Again, analogously to the corresponding string operations.
https://docs.python.org/3/library/stdtypes.html#typesseq

Important concept: immutability

What if I want to change a letter in my string?

Try and assign a different string to a subsequence of a string.

```
1 mystr = 'goat'
2 mystr[0] = 'b'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-cf531ebc1ce4> in <module>()
      1 mystr = 'goat'
----> 2 mystr[0] = 'b'

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

We get an error because strings are **immutable**.
We can't change the value of an *existing* string.

Important concept: immutability

What if I want to change a letter in my string?

```
mystr = 'goat'
mystr = 'b'+mystr[1:]
mystr
```

```
'boat'
```

This avoids the error we saw before because it changes the value of the variable `mystr`, rather than trying to change the contents of a string.

List Methods

Lists supply a certain set of methods:

`list.append(x)`: adds `x` to the end of the list

`list.extend(L2)`: adds list `L2` to the end of another list (like concatenation)

`list.sort()`: sort the elements of the list

`list.remove(x)`: removes from the list the first element equal to `x`.

`list.pop()`: removes the last element of the list and returns that element.

`list.append()` and `list.extend()`

```
animals = ['cat', 'dog', 'goat', 'bird']
animals.append('unicorn')
animals
```

```
['cat', 'dog', 'goat', 'bird', 'unicorn']
```

We call list methods with dot notation. These are **methods** supported by certain **objects**.

```
fibonacci = [0,1,1,2,3,5,8]
fibonacci.append([13,21])
fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, [13, 21]]
```

Warning: `list.append()` adds its argument as the last element of a list! Use `list.extend()` to concatenate to the end of the list!

```
fibonacci = [0,1,1,2,3,5,8]
fibonacci.extend([13, 21])
fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Note: all of these list methods act upon the list that calls the method. These methods don't return the new list, they alter the list on which we call them.

`list.sort()` and `sorted()`

```
animals = ['cat', 'dog', 'goat', 'bird']
animals.sort()
animals
```

```
['bird', 'cat', 'dog', 'goat']
```

```
mixed = [1, 'two', 3.0, [4,5]]
mixed.sort()
mixed
```

```
[1, 3.0, [4, 5], 'two']
```

```
animals = ['cat', 'dog', 'goat', 'bird']
sorted_animals = sorted(animals)
sorted_animals
```

```
['bird', 'cat', 'dog', 'goat']
```

```
animals
```

```
['cat', 'dog', 'goat', 'bird']
```

`list.sort()` sorts the list **in place**. See documentation for how Python sorts data of different types.

If I don't want to sort a list in place, the `sorted()` command returns a sorted version of the list, leaving its argument unchanged.

Removing elements: `list.pop()`

```
1 animals = ['cat', 'dog', 'goat', 'bird']
2 animals.pop()
```

'bird'

```
1 animals
```

['cat', 'dog', 'goat']

```
1 fibonacci = [0,1,1,2,3,5,8]
2 fibonacci.pop(3)
```

2

```
1 fibonacci
```

[0, 1, 1, 3, 5, 8]

`list.pop()` removes the last element from the list and returns that element.

`list.pop()` takes an **optional argument**, which indexes into the list and removes and returns the indexed element

Again, this method alters the list itself, rather than returning an altered list.

Removing elements: `list.remove()`

```
1 animals = ['cat', 'dog', 'goat', 'bird']
2 animals.remove('cat')
3 animals
```

```
['dog', 'goat', 'bird']
```

```
1 numbers = [0,1,2,3,1,2,3,2,3]
2 numbers.remove(2)
3 numbers
```

```
[0, 1, 3, 1, 2, 3, 2, 3]
```

`list.remove(x)` removes the first instance of `x` in the list.

```
1 numbers.remove(4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-160-6d289ee6c03d> in <module>()
----> 1 numbers.remove(4)

ValueError: list.remove(x): x not in list
```

Raises a `ValueError` if no such element exists.

Map

Example: suppose I want to square every element of a list.

```
def square_all(t):
    res = []
    for elmt in t:
        res.append(elmt**2)
    return res

square_all(range(10))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
fibonacci = [0,1,1,2,3,5,8,13,21]
square_all(fibonacci)
```

```
[0, 1, 1, 4, 9, 25, 64, 169, 441]
```

```
fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21]
```

This function takes a list `t`, and creates a new list `res`, which consists of the squares of the elements of `t`.

This kind of operation, in which we apply a function to each element of a list, is called a **map** operation.

Note: unlike the list methods in the previous slides, this function creates a new list, and doesn't alter the argument.

Map with list comprehensions

Basic pattern: `[f(x) for x in mylist]` creates a new list, whose elements are the elements of `mylist`, each with function `f` applied.

```
zero2nine = range(10)
[x**2 for x in zero2nine]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note: the function `f` must actually return something!

```
animals = ['cat', 'dog', 'goat', 'bird']
[s.upper() for s in animals]
```

```
['CAT', 'DOG', 'GOAT', 'BIRD']
```

List comprehensions are a special pattern supplied by Python. They're one of the features of Python that makes it appealing. Very expressive way to write operations!

Filter

Example: I want to remove all even numbers from a list.

```
def remove_even(t):
    res = []
    for elmt in t:
        if elmt % 2 == 0:
            continue
        else: # elmt is odd.
            res.append(elmt)
    return res

remove_even(range(10))
```

```
[1, 3, 5, 7, 9]
```

This function takes a list `t`, and creates a new list `res`, which contains only the odd elements of `t`.

This kind of operation, in which we keep only the elements of a list that satisfy some condition, is called a **filter** operation.

```
fibonacci = [0,1,1,2,3,5,8,13,21]
remove_even(fibonacci)
```

```
[1, 1, 3, 5, 13, 21]
```

```
fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21]
```

Note: again, this function creates a new list, and doesn't alter the argument.

Filter with list comprehensions

```
fibonacci = [0,1,1,2,3,5,8,13,21]
[x for x in fibonacci if x % 2 ==1]
```

```
[1, 1, 3, 5, 13, 21]
```

Basic pattern:
`[x for x in mylist if boolean_expr]`
creates a new list of all and only the elements of `mylist` that satisfy `boolean_expr`.

```
animals = ['cat', 'dog', 'goat', 'bird']
[x.upper() for x in animals if 'o' in x[1]]
```

```
['DOG', 'GOAT']
```

Can combine filter and map to apply a function to only the elements that pass the filter.

```
[x for x in animals if len(x)==5]
```

```
[]
```

Lists and strings

Lists and strings are both sequences, but they aren't quite the same...

```
goatstr = 'goat'
goatlist = list(goatstr)
goatlist
```

```
['g', 'o', 'a', 't']
```

`str.split()` turns a string into a list of strings, splitting the string on its argument, called the **delimiter**.

```
wittgenstein = 'Die Welt ist alles was der Fall ist.'
t = wittgenstein.split(' ')
t
```

```
['Die', 'Welt', 'ist', 'alles', 'was', 'der', 'Fall', 'ist.']
```

`str.join()` is like the inverse of `str.split()`. It takes a list of strings and joins them into a single string.

```
delim = ' '
delim.join(t)
```

```
'Die Welt ist alles was der Fall ist.'
```

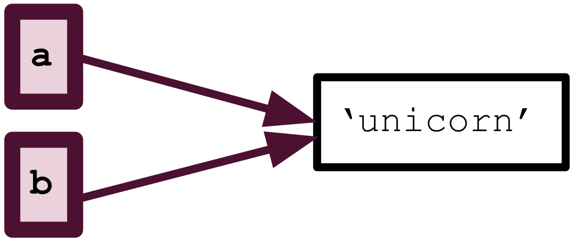
Equivalent vs identical objects

```
a = 'unicorn'
b = 'unicorn'
```

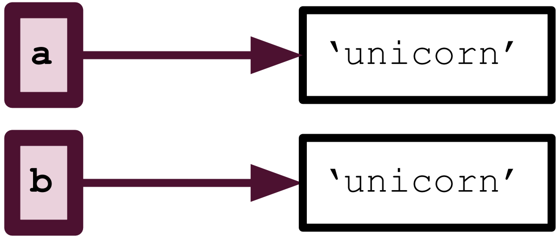
Question: are `a` and `b` the same?

Well, what do we mean by "the same"?

Possibility 1:
`a` and `b` both 'point to' the *same* object.



Possibility 2:
`a` and `b` 'point to' different objects, both objects have *same value*.



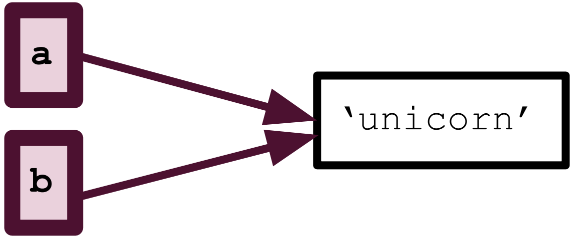
Equivalent vs identical objects

```
a = 'unicorn'
b = 'unicorn'
```

Question: are `a` and `b` the same?

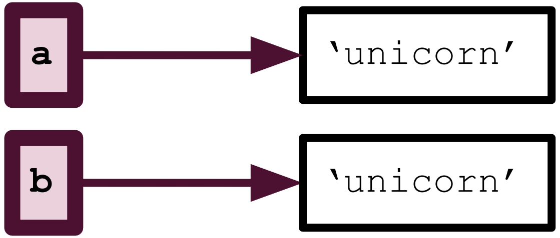
Well, what do we mean by "the same"?

Possibility 1:
`a` and `b` both 'point to' the *same* object.



In this case, we say that `a` and `b` are **identical**

Possibility 2:
`a` and `b` 'point to' different objects, both objects have *same value*.



In this case, we say that `a` and `b` are **equivalent**

Equivalent vs identical objects

```
a = 'unicorn'
b = 'unicorn'
```

```
a == b
```

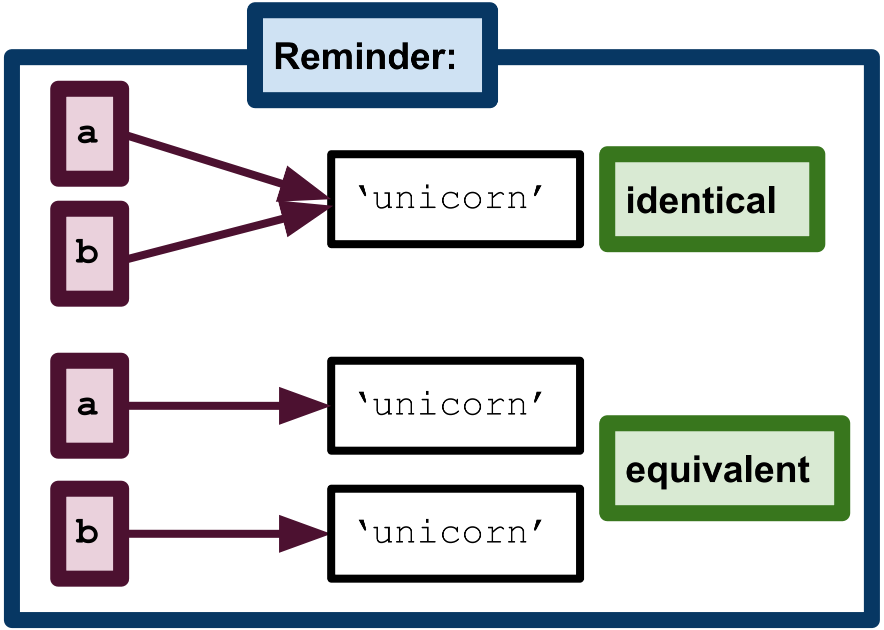
True

```
a is b
```

True

`==` tests if two variables are **equivalent**.
`is` tests if two variables are **identical**.

Strings are immutable, so Python only creates one copy of the string `'unicorn'`, and both `a` and `b` point to it. So they are equivalent **and** identical.



Equivalent vs identical objects

```
a = [1,2,3]
b = [1,2,3]
a == b
```

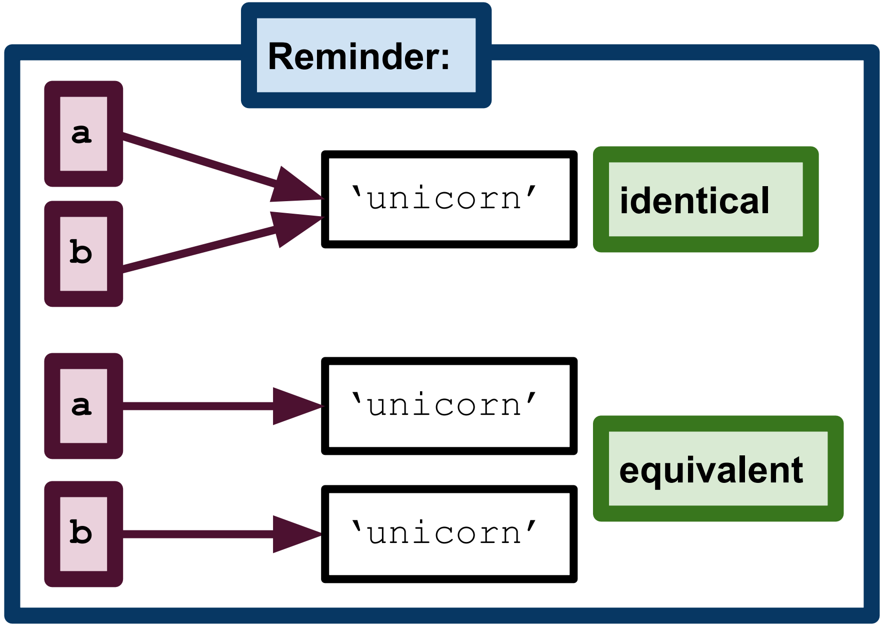
True

```
a is b
```

False

`==` tests if two variables are **equivalent**.
`is` tests if two variables are **identical**.

Lists are mutable, so Python creates different copies for `a` and `b`. So they are equivalent but **not** identical.

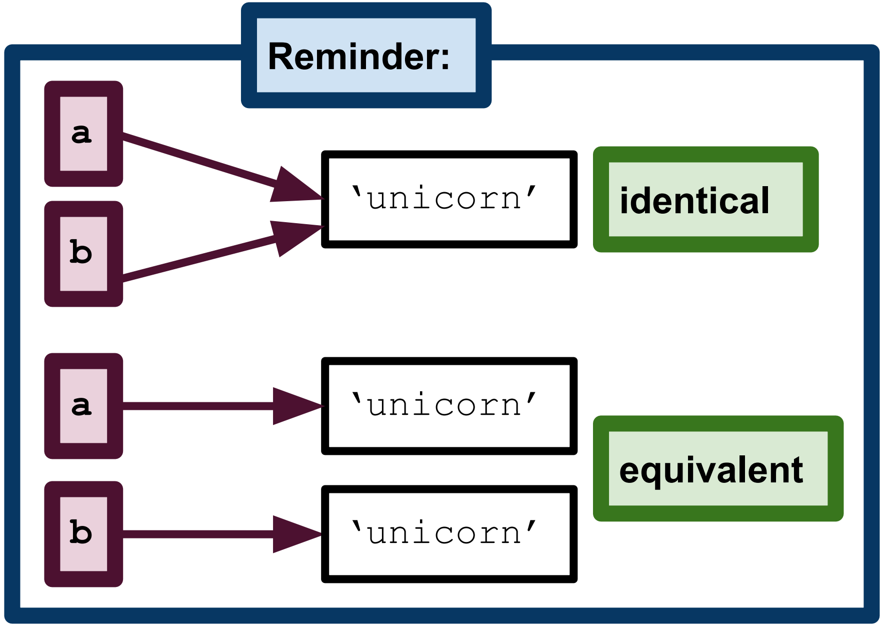


Equivalent vs identical objects: reference

```
a = [1,2,3]
b = a
a is b
```

`==` tests if two variables are **equivalent**.
`is` tests if two variables are **identical**.

Question: will this evaluate to `True` or `False`?



Equivalent vs identical objects: reference

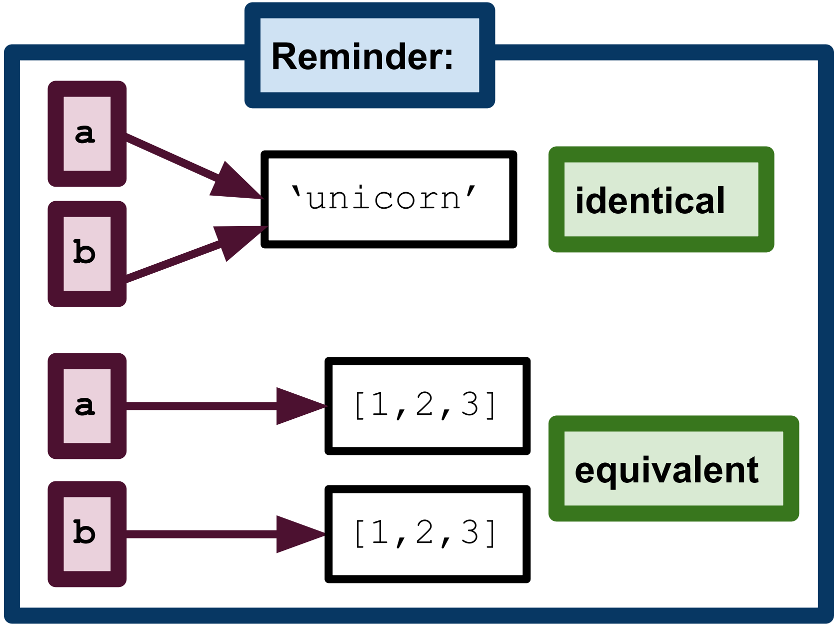
```
a = [1,2,3]
b = a
a is b
```

```
True
```

`==` tests if two variables are **equivalent**.
`is` tests if two variables are **identical**.

Answer: evaluates to `True`, because assignment changes the **reference** of a variable.

Reference of a variable is the value to which it “points”, like on the right.



An object that has more than one reference (i.e., more than one “name”) is called **aliased**. So, on the right, `'unicorn'` is aliased.

Equivalent vs identical objects: reference

```
1 a = [1,2,3]
2 b = a
3 b[-1] = 42
4 b
```

```
[1, 2, 42]
```

```
1 a[-1]
```

Warning: Aliased mutable objects can sometimes cause unexpected behavior.

Question: what should this evaluate to?

Equivalent vs identical objects: reference

```
a = [1,2,3]
b = a
b[-1] = 42
b
```

```
[1, 2, 42]
```

```
a[-1]
```

```
42
```

Warning: Aliased mutable objects can sometimes cause unexpected behavior.

Question: what should this evaluate to?

Answer: when we changed the last element of `b`, we changed the object referenced by both `a` and `b`.

Pass-by-reference vs pass-by-value

```
def make_end_42(t):
    # Change the last element of
    # list t to be 42.
    t[-1] = 42

a = [1,2,3]
make_end_42(a)
a
```

```
[1, 2, 42]
```

When you pass an object to a function, the function gets a reference to that object. So changes that we make inside the function are also true outside. This is called **pass-by-reference**, because the function gets a reference to its argument.

Pass-by-reference vs pass-by-value

```
def wrong_make_end_42(t):
    # Change the last element of
    # list t to be 42, incorrectly.
    t = t[:-1] # delete the last element.
    t.append(42)

a = [1,2,3]
wrong_make_end_42(a)
a
```

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

When we make the assignment to `t`, we create a new list, and the reference of `t` is changed, so it no longer points to the list that we passed to the function!

Moral of the story: be careful when working with mutable objects, especially when you are trying to modify objects in place. Often, it's better to just write a function that modifies a list and returns the modified list!

Dictionaries (and Tuples)

Two more fundamental built-in data structures

Dictionaries

- Python dictionaries generalize lists
- Allow indexing by arbitrary immutable objects rather than integers
- Fast lookup and retrieval
- https://docs.python.org/3/tutorial/datastructures.html#dictionaries

Tuples

- Similar to a list, in that it is a sequence of values
- But unlike lists, tuples are immutable
- https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

Generalized lists: Python `dict()`

Python dictionary generalizes lists

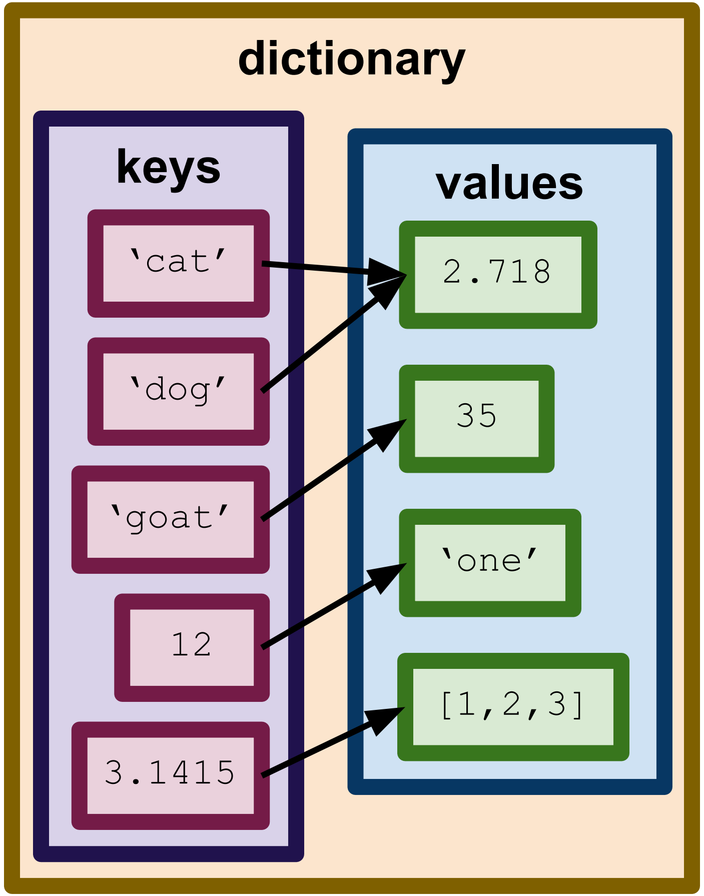
- `list()`: indexed by integers
- `dict()`: indexed by (almost) any data type

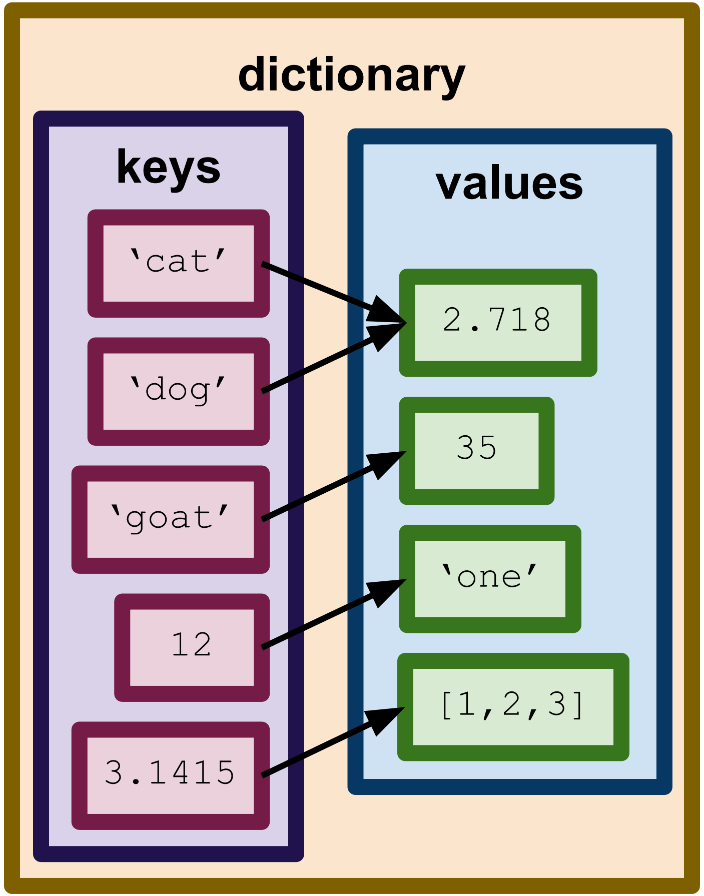
Dictionary contains:

- a set of indices, called **keys**
- A set of values (called **values**, shockingly)

Each key associated with one (and only one) value

- **key-value pairs**, sometimes called **items**
- Like a function f: keys -> values



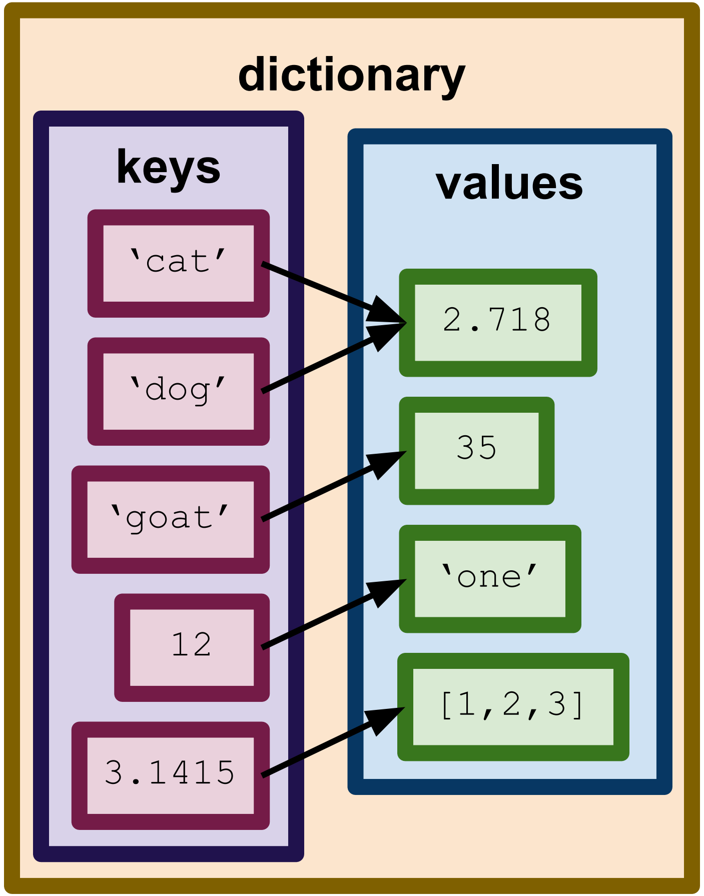


Dictionary **maps** keys to values.

E.g., `'cat'` mapped to the float 2.718

Of course, the dictionary at the left is kind of silly. In practice, keys are often all of the same type, because they all represent a similar kind of object

Example: might use a dictionary to map UMich unique names to people




```
1 example_dict['goat']
```
35

```
1 example_dict['cat']
```
2.718

```
1 example_dict['dog']
```
2.718

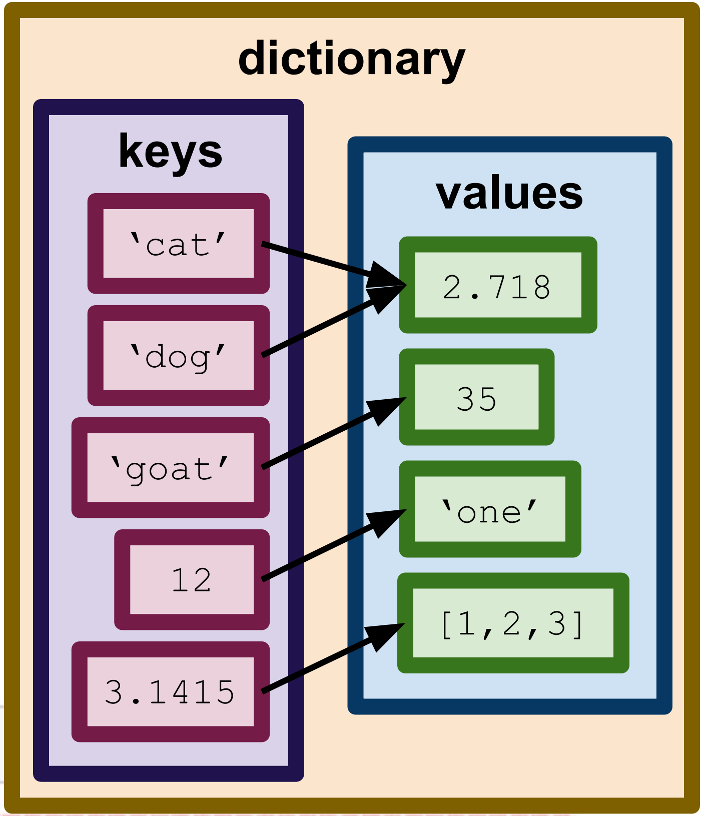
```
1 example_dict[3.1415]
```
[1, 2, 3]

```
1 example_dict[12]
```
'one'

Access the value associated to key `x` by `dictionary[x]`.

Attempting to access the value associated to a non-existent key results in a `KeyError`, an error that Python supplies specifically for this situation.

Observe that bird is not a key in this dictionary, so when we try to index with it, we get an error.




```
1 example_dict['bird']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-057007a28b5b> in <module>()
----> 1 example_dict['bird']

KeyError: 'bird'
```

Creating and populating a dictionary

Example: University of Michuges IT wants to store the correspondence between the usernames (UM IDs) of students to their actual names. A dictionary is a very natural data structure for this.

```
umid2name = dict()
umid2name['aeinstein'] = 'Albert Einstein'
umid2name['kyfan'] = 'Ky Fan'
umid2name['enoether'] = 'Emmy Noether'
umid2name['cshannon'] = 'Claude Shannon'
```

```
umid2name['cshannon']
```

```
'Claude Shannon'
```

```
umid2name['enoether']
```

```
'Emmy Noether'
```

```
umid2name['enoether'] = 'Amalie Emmy Noether'
umid2name['enoether']
```

```
'Amalie Emmy Noether'
```

Creating and populating a dictionary

Create an empty dictionary (i.e., a dictionary with no key-value pairs stored in it. This should look familiar, since it is very similar to list creation.

```
umid2name = dict()
umid2name['aeinstein'] = 'Albert Einstein'
umid2name['kyfan'] = 'Ky Fan'
umid2name['enoether'] = 'Emmy Noether'
umid2name['cshannon'] = 'Claude Shannon'
```

```
umid2name['cshannon']
```

```
'Claude Shannon'
```

```
umid2name['enoether']
```

```
'Emmy Noether'
```

```
umid2name['enoether'] = 'Amalie Emmy Noether'
umid2name['enoether']
```

```
'Amalie Emmy Noether'
```

Creating and populating a dictionary

Populate the dictionary. We are adding four key-value pairs, corresponding to four users in the system.

```
umid2name = dict()
umid2name['aeinstein'] = 'Albert Einstein'
umid2name['kyfan'] = 'Ky Fan'
umid2name['enoether'] = 'Emmy Noether'
umid2name['cshannon'] = 'Claude Shannon'
```

```
umid2name['cshannon']
```

```
'Claude Shannon'
```

```
umid2name['enoether']
```

```
'Emmy Noether'
```

```
umid2name['enoether'] = 'Amalie Emmy Noether'
umid2name['enoether']
```

```
'Amalie Emmy Noether'
```

Creating and populating a dictionary

```
umid2name = dict()
umid2name['aeinstein'] = 'Albert Einstein'
umid2name['kyfan'] = 'Ky Fan'
umid2name['enoether'] = 'Emmy Noether'
umid2name['cshannon'] = 'Claude Shannon'
```

```
umid2name['cshannon']
```

```
'Claude Shannon'
```

```
umid2name['enoether']
```

```
'Emmy Noether'
```

Retrieve the value associated with a key. This is called **lookup**.

```
umid2name['enoether'] = 'Amalie Emmy Noether'
umid2name['enoether']
```

```
'Amalie Emmy Noether'
```

Creating and populating a dictionary

```
umid2name = dict()
umid2name['aeinstein'] = 'Albert Einstein'
umid2name['kyfan'] = 'Ky Fan'
umid2name['enoether'] = 'Emmy Noether'
umid2name['cshannon'] = 'Claude Shannon'
```

```
umid2name['cshannon']
```

```
'Claude Shannon'
```

```
umid2name['enoether']
```

```
'Emmy Noether'
```

Emmy Noether’s actual legal name was Amalie Emmy Noether, so we have to update her record. Note that updating is syntactically the same as initial population of the dictionary.

```
umid2name['enoether'] = 'Amalie Emmy Noether
umid2name['enoether']
```

```
Amalie Emmy Noether'
```

Displaying Items

Printing a dictionary lists its **items** (key-value pairs), in this rather odd format...

```
example_dict
```

```
{3.1415: [1, 2, 3], 12: 'one', 'cat': 2.718, 'dog': 2.718, 'goat': 35}
```

```
umid2name
```

```
{'aeinstein': 'Albert Einstein',
 'cshannon': 'Claude Shannon',
 'enoether': 'Amalie Emmy Noether',
 'kyfan': 'Ky Fan'}
```

...but I can use that format to create a new dictionary.

```
umid2name = {'aeinstein': 'Albert Einstein',
  'cshannon': 'Claude Shannon',
  'enoether': 'Amalie Emmy Noether',
  'kyfan': 'Ky Fan'}
```

```
umid2name['kyfan']
```

```
'Ky Fan'
```

Note: the order in which items are printed isn't always the same, and (usually) isn't predictable. This is due to how dictionaries are stored in memory. More on this soon.

Dictionaries have a length

```
1 umid2name
```

```
{'aeinstein': 'Albert Einstein',
 'cshannon': 'Claude Shannon',
 'enoether': 'Amalie Emmy Noether',
 'kyfan': 'Ky Fan'}
```

Length of a dictionary is just the number of items.

```
1 len(umid2name)
```

```
4
```

Empty dictionary has length 0.

```
1 d = dict()
2 len(d)
```

```
0
```

Note: we said earlier than all sequence objects support the length operation. But there exist objects that **aren't** sequences that also have this attribute.

Checking set membership

```
1  umid2name
```

```
{'aeinstein': 'Albert Einstein',
 'cshannon': 'Claude Shannon',
 'enoether': 'Amalie Emmy Noether',
 'kyfan': 'Ky Fan'}
```

```
1  'akolmogorov' in umid2name
```

```
False
```

```
1  'enoether' in umid2name
```

```
True
```

Suppose a new student, Andrey Kolmogorov is enrolling at UMish. We need to give him a unique name, but we want to make sure we aren't assigning a name that's already taken.

Dictionaries support checking whether or not an element is present **as a key**, similar to how lists support checking whether or not an element is present in the list.

Checking set membership: fast and slow

```python
from random import randint
listlen = 1000000
list_of_numbers = listlen*[0]
dict_of_numbers = dict()
for i in range(listlen):
    n = randint(1000000,9999999)
    list_of_numbers[i] = n
    dict_of_numbers[n] = 1
```

```python
8675309 in list_of_numbers
```

```
False
```

```python
1240893 in list_of_numbers
```

```
True
```

```python
8675309 in dict_of_numbers
```

```
False
```

```python
1240893 in dict_of_numbers
```

```
True
```

Lists and dictionaries provide our first example of how certain **data structures** are better for certain tasks than others.

Example: I have a large collection of phone numbers, and I need to check whether or not a given number appears in the collection. Both dictionaries and lists support **membership checks** of this sort, but it turns out that dictionaries are much better suited to the job.

Checking set membership: fast and slow

```
from random import randint
listlen = 1000000
list_of_numbers = listlen*[0]
dict_of_numbers = dict()
for i in range(listlen):
    n = randint(1000000,9999999)
    list_of_numbers[i] = n
    dict_of_numbers[n] = 1
```

This block of code generates 1000000 random “phone numbers”, and creates
(1) a list of all the numbers and
(2) a dictionary whose keys are all the numbers.

```
8675309 in list_of_numbers
```

```
False
```

```
1240893 in list_of_numbers
```

```
True
```

```
8675309 in dict_of_numbers
```

```
False
```

```
1240893 in dict_of_numbers
```

```
True
```

Checking set membership: fast and slow

```python
from random import randint
listlen = 1000000
list_of_numbers = listlen*[0]
dict_of_numbers = dict()
for i in range(listlen):
    n = randint(1000000,9999999)
    list_of_numbers[i] = n
    dict_of_numbers[n] = 1
```

```python
8675309 in list_of_numbers
```

```
False
```

```python
1240893 in list_of_numbers
```

```
True
```

```python
8675309 in dict_of_numbers
```

```
False
```

```python
1240893 in dict_of_numbers
```

```
True
```

The `random` module supports a bunch of random number generation operations. We'll see more on this later in the course. https://docs.python.org/3/library/random.html

Checking set membership: fast and slow

```
from random import randint
listlen = 1000000
list_of_numbers = listlen*[0]
dict_of_numbers = dict()
for i in range(listlen):
    n = randint(1000000,9999999)
    list_of_numbers[i] = n
    dict_of_numbers[n] = 1
```

```
8675309 in list_of_numbers
```

```
False
```

```
1240893 in list_of_numbers
```

```
True
```

```
8675309 in dict_of_numbers
```

```
False
```

```
1240893 in dict_of_numbers
```

```
True
```

Initialize a list (of all zeros) and an empty dictionary.

Checking set membership: fast and slow

```python
from random import randint
listlen = 1000000
list_of_numbers = listlen*[0]
dict_of_numbers = dict()
for i in range(listlen):
    n = randint(1000000,9999999)
    list_of_numbers[i] = n
    dict_of_numbers[n] = 1
```

Generate `listlen` random numbers, writing them to both the list and the dictionary.

```python
8675309 in list_of_numbers
```

```
False
```

```python
1240893 in list_of_numbers
```

```
True
```

```python
8675309 in dict_of_numbers
```

```
False
```

```python
1240893 in dict_of_numbers
```

```
True
```

Checking set membership: fast and slow

```
from random import randint
listlen = 1000000
list_of_numbers = listlen*[0]
dict_of_numbers = dict()
for i in range(listlen):
    n = randint(1000000,9999999)
    list_of_numbers[i] = n
    dict_of_numbers[n] = 1
```

```
8675309 in list_of_numbers
```

False

```
1240893 in list_of_numbers
```

True

This is slow.

```
8675309 in dict_of_numbers
```

False

```
1240893 in dict_of_numbers
```

True

This is fast.

Checking set membership: fast and slow

```
import time
start_time = time.time()
8675309 in list_of_numbers
time.time() - start_time
```

0.10922789573669434

```
start_time = time.time()
8675309 in dict_of_numbers
time.time() - start_time
```

0.0002219676971435547

Let’s get a more quantitative look at the difference in speed between lists and dicts.

The `time` module supports accessing the system clock, timing functions, and related operations.
https://docs.python.org/3/library/time.html
Timing parts of your program to find where performance can be improved is called **profiling** your code. Python provides some built-in tools for more profiling, which we’ll discuss later in the course, if time allows.
https://docs.python.org/3/library/profile.html

Checking set membership: fast and slow

```
import time
start_time = time.time()
8675309 in list_of_numbers
time.time() - start_time
```

```
0.1092278957369434
```

To see how long an operation takes, look at what time it is, perform the operation, and then look at what time it is again. The time difference is how long it took to perform the operation.

```
start_time = time.time()
8675309 in dict_of_numbers
time.time() - start_time
```

```
0.0002219676971435547
```

Warning: this can be influenced by other processes running on your computer. See documentation for ways to mitigate that inaccuracy.

Checking set membership: fast and slow

```
import time
start_time = time.time()
8675309 in list_of_numbers
time.time() - start_time
```

```
0.10922789573669434
```

```
start_time = time.time()
8675309 in dict_of_numbers
time.time() - start_time
```

```
0.00022196769714355469
```

Checking membership in the dictionary is orders of magnitude faster! Why should that be?

Checking set membership: fast and slow

```
import time
start_time = time.time()
8675309 in list_of_numbers
time.time() - start_time
```

0.10922789573669434

```
start_time = time.time()
8675309 in dict_of_numbers
time.time() - start_time
```

0.00021967697143554 7

The time difference is due to how the in operation is implemented for lists and dictionaries.

Python compares `x` against each element in the list until it finds a match or hits the end of the list. So this takes time **linear** in the length of the list.

Python uses a **hash table**. For now, it suffices to know that this lets us check if `x` is in the dictionary in (almost) the same amount of time, regardless of how many items are in the dictionary.