STATS 507
Data Analysis in Python

Week 3: Dictionaries (again), Tuples, Files, Exceptions, and Modules

adapted from slides by Keith Levin

Checking set membership: fast and slow

```
1 from random import randint
2 listlen = 1000000
3 list_of_numbers = listlen*[0]
4 dict_of_numbers = dict()
5 for i in range(listlen):
6     n = randint(1000000,9999999)
7     list_of_numbers[i] = n
8     dict_of_numbers[n] = 1
```

```
1 8675309 in list_of_numbers
```

```
False
```

```
1 1240893 in list_of_numbers
```

```
True
```

This is slow.

```
1 8675309 in dict_of_numbers
```

```
False
```

```
1 1240893 in dict_of_numbers
```

```
True
```

This is fast.

Checking set membership: fast and slow

```
1 import time
2 start_time = time.time()
3 8675309 in list_of_numbers
4 time.time() - start_time
```

```
0.10922789573669434
```

```
1 start_time = time.time()
2 8675309 in dict_of_numbers
3 time.time() - start_time
```

```
0.00022196769714355547
```

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

Crash course: hash tables

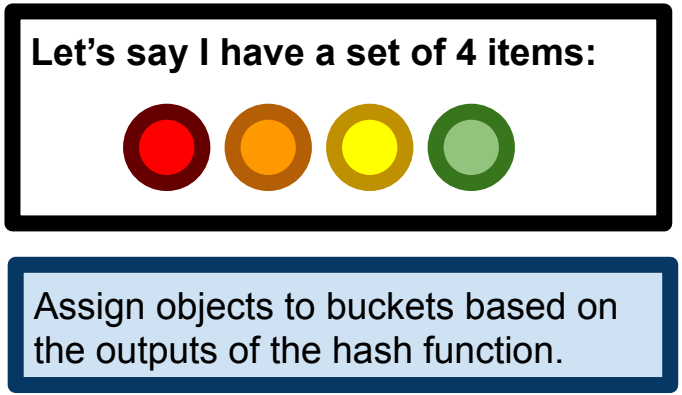
Universe of objects

Let's say I have a set of 4 items:

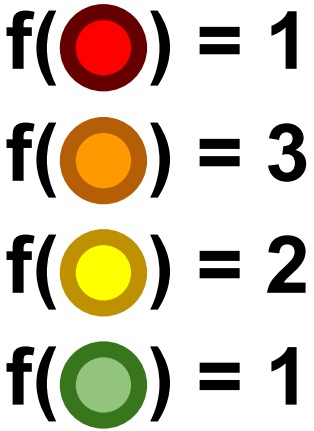
I want to find a way to know **quickly** whether or not an item is in this set.

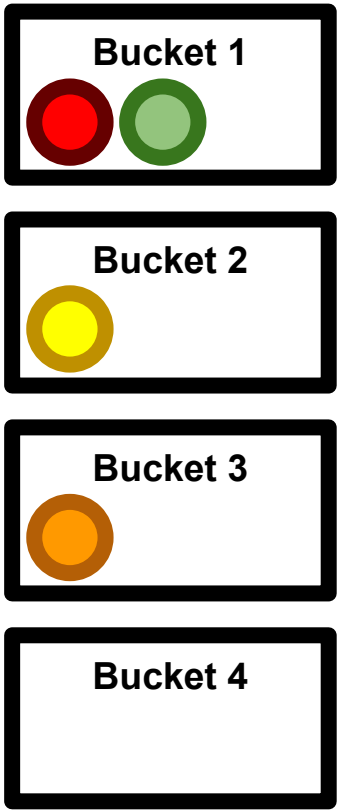
Crash course: hash tables

Hash function **f** maps objects to “buckets”



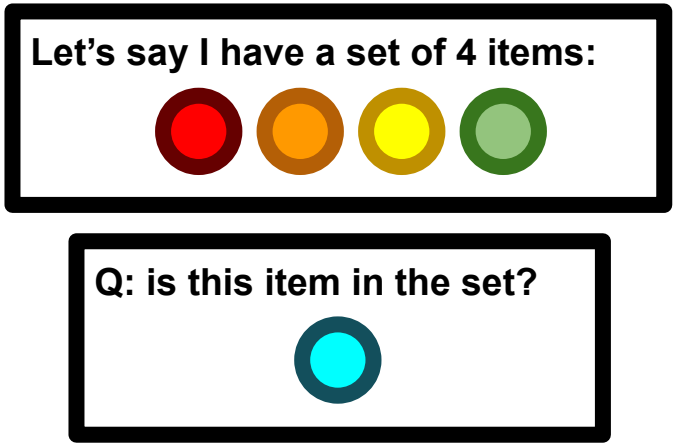
Assign objects to buckets based on the outputs of the hash function.

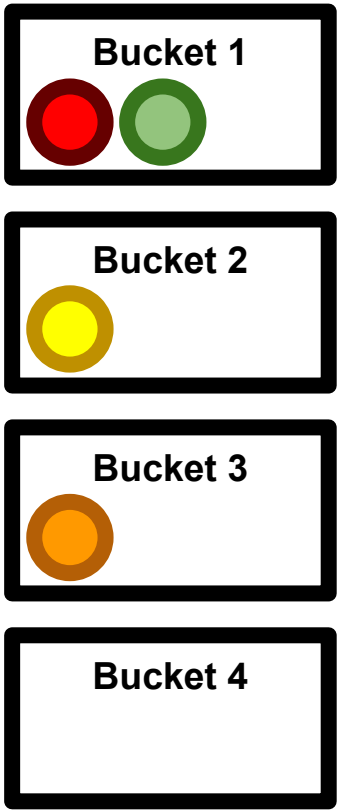




Crash course: hash tables

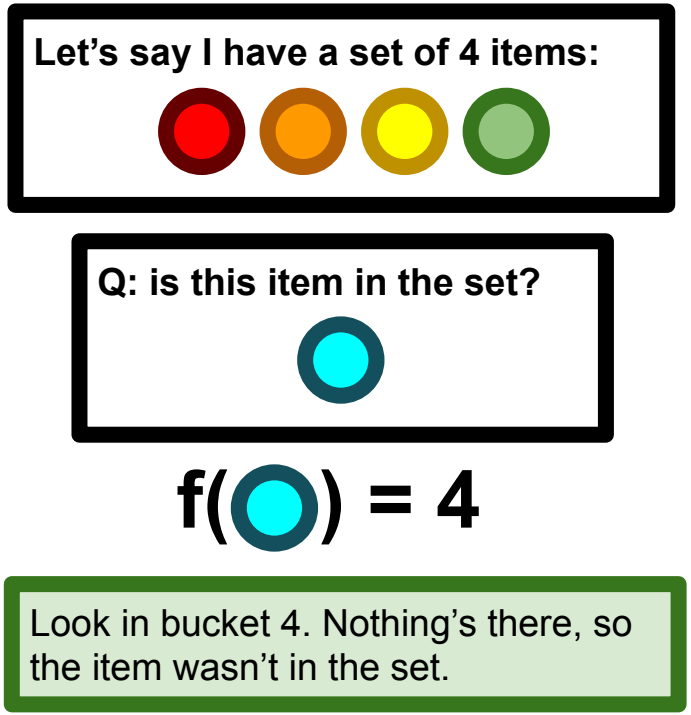
Hash function maps objects to “buckets”

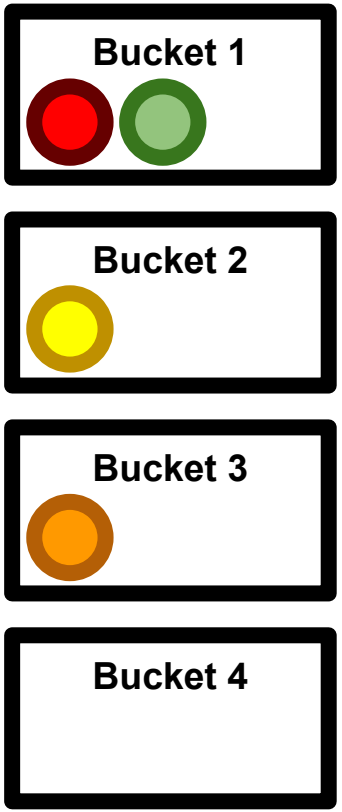




Crash course: hash tables

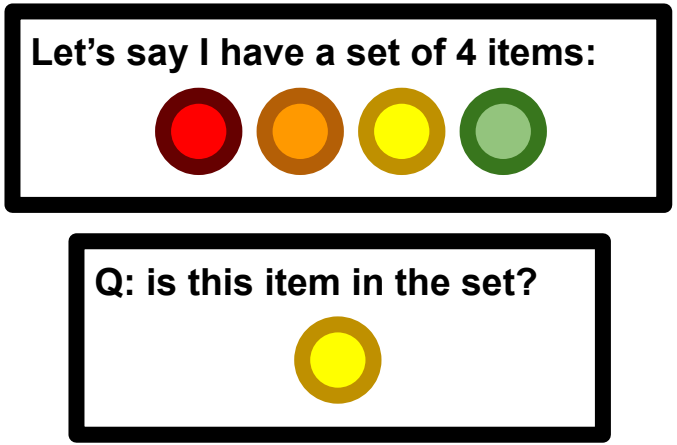
Hash function maps objects to “buckets”

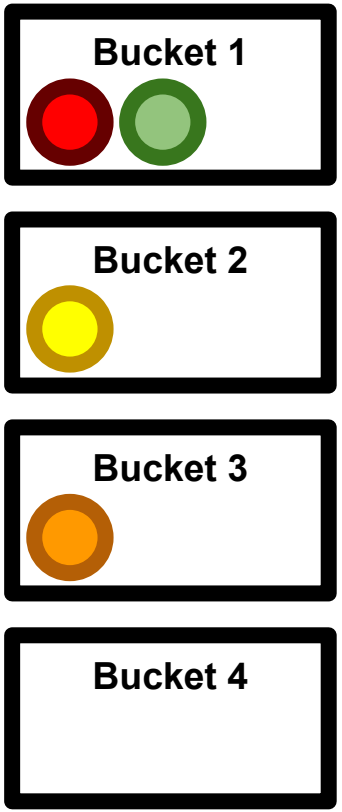




Crash course: hash tables

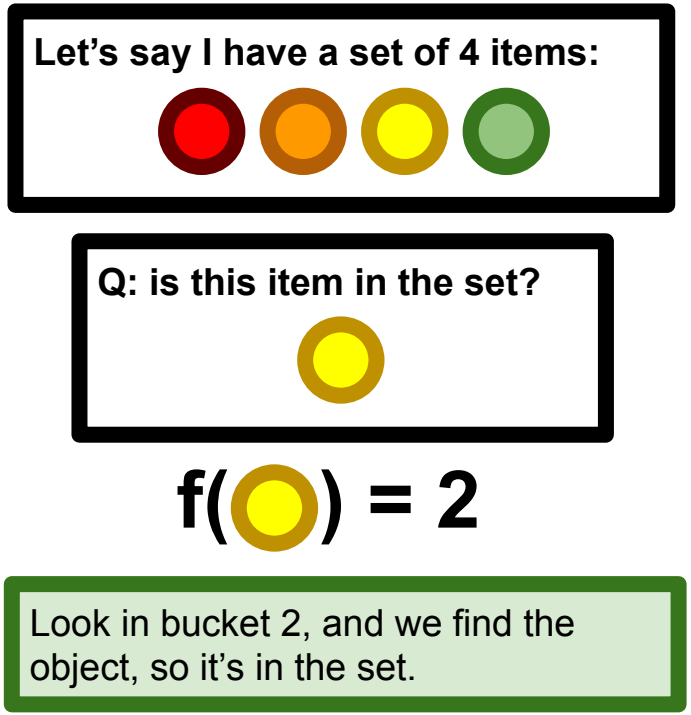
Hash function maps objects to “buckets”

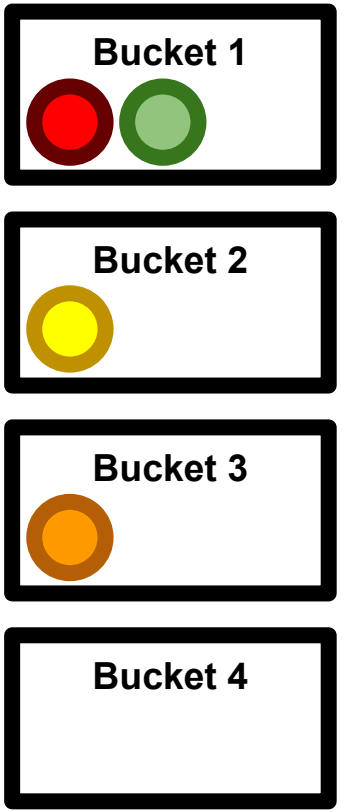




Crash course: hash tables

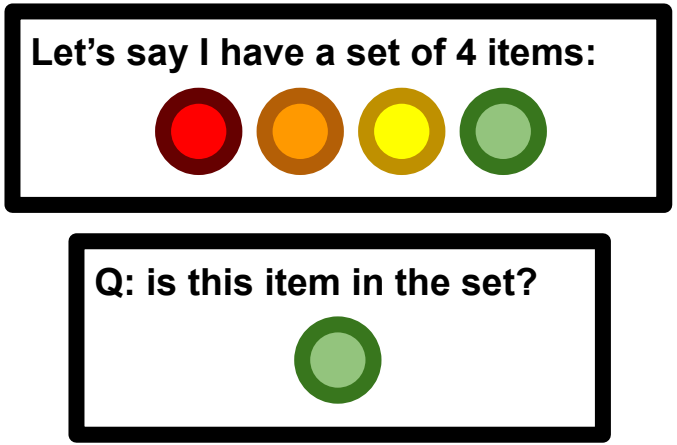
Hash function maps objects to “buckets”

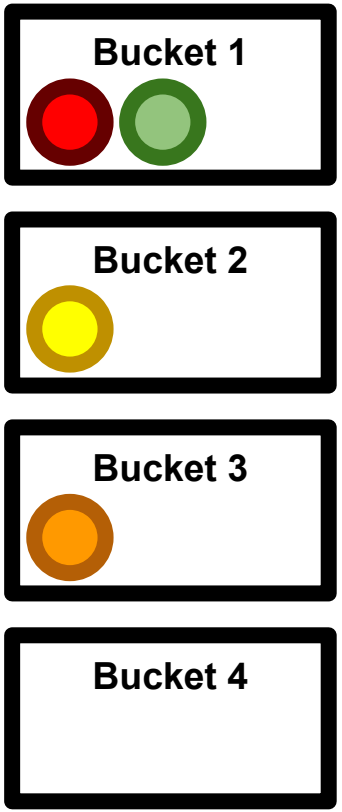




Crash course: hash tables

Hash function maps objects to “buckets”





Crash course: hash tables

When more than one object falls in the same bucket, we call it a **hash collision**.

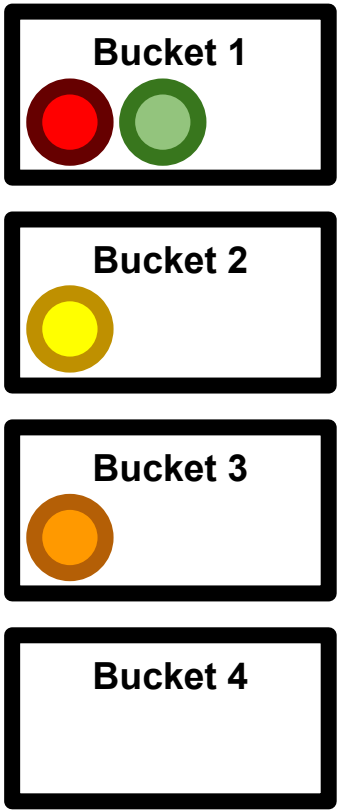
Hash function maps objects to “buckets”

Let’s say I have a set of 4 items:

Q: is this item in the set?

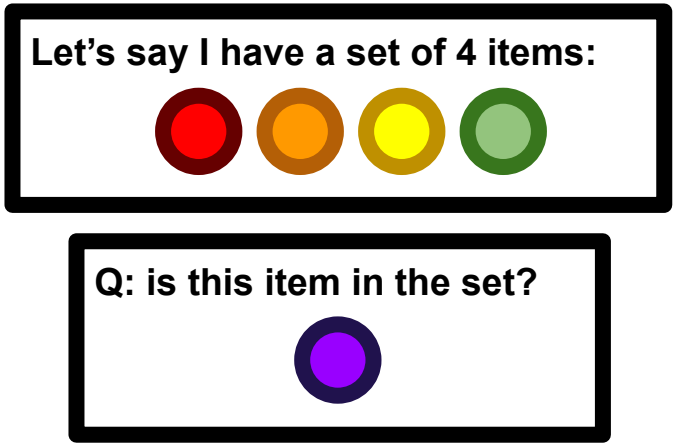
f( ) = 1

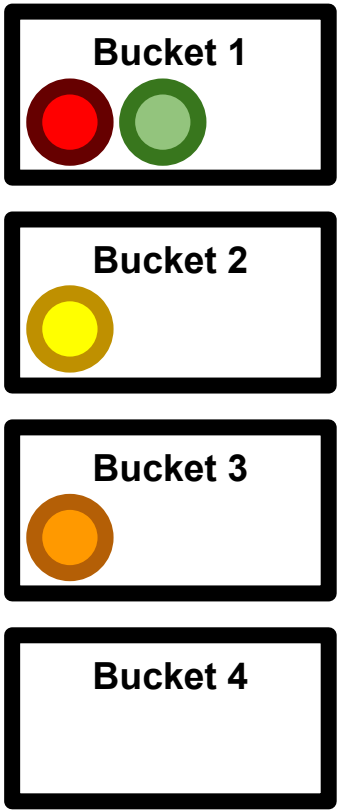
Look in bucket 1, and there’s more than one thing. Compare against each of them, eventually find a match.



Crash course: hash tables

Hash function maps objects to “buckets”





Crash course: hash tables

Worst possible case: have to check everything in the bucket only to conclude there's no match.

Hash function maps objects to "buckets"

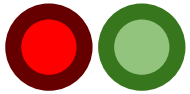
Let's say I have a set of 4 items:

Q: is this item in the set?

f( ) = 1

Look in bucket 1, and there's more than one thing. Compare against each of them, no match, so it's not in the set.

Bucket 1



Bucket 2

Bucket 3

Bucket 4

Crash course: hash tables

Hash function maps objects to “buckets”

Key point: hash table lets us avoid comparing against every object in the set (provided we pick a good hash function that has few collisions)

More information:

https://en.wikipedia.org/wiki/Hash_table

https://en.wikipedia.org/wiki/Hash_function

For the purposes of this course, it suffices to know that dictionaries (and the related **set** object, which we’ll see soon), have faster membership checking than lists because they use hash tables.

Common pattern: dictionary as counter

Example: counting word frequencies

Naïve idea: keep one variable to keep track of each word

We're gonna need a lot of variables!

Better idea: use a dictionary, keep track of only the words we see

```
1 wdcounts = dict()
2 for w in list_of_words:
3     wdcounts[w] += 1
```

This code as written won't work! It's your job in one of your homework problems to flesh this out. You may find it useful to read about the `dict.get()` method: https://docs.python.org/3/library/stdtypes.html#dict.get

Traversing a dictionary

Suppose I have a dictionary representing word counts…

...and now I want to display the counts for each word.

```
1 for w in wdcnt:
2     print(w, wdcnt[w])
```

```
half 3
a 3
league 3
onward 1
all 1
in 1
the 2
valley 1
of 1
death 1
rode 1
six 1
hundred 1
```

Traversing a dictionary yields the keys, in no particular order. Typically, you'll get them in the order they were added, but this is not guaranteed, so don't rely on it.

This kind of traversal is, once again, a very common pattern when dealing with dictionaries. Dictionaries support iteration over their keys. They, like sequences, are **iterators**. We'll see more of this as the course continues.
https://docs.python.org/dev/library/stdtypes.html#iterator-types

Common Pattern: Reverse Lookup and Inversion

Returning to our example, what if I want to map a (real) name to a uniqname?

E.g., I want to look up Emmy Noether's username from her real name

```
1 umid2name
```

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

```
1 name2umid = dict()
2 for uname in umid2name:
3     truename = umid2name[uname]
4     name2umid[truename] = uname
5 name2umid
```

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

The keys of `umid2name` are the values of `name2umid` and vice versa. We say that `name2umid` is the **reverse lookup** table (or the **inverse**) for `umid2name`.

Common Pattern: Reverse Lookup and Inversion

Returning to our example, what if I want to map a (real) name to a uniqname?

E.g., I want to look up Emmy Noether's username from her real name

```
1 umid2name
```

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

```
1 name2umid = dict()
2 for uname in umid2name:
3     truename = umid2name[uname]
4     name2umid[truename] = uname
5 name2umid
```

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

The keys of `umid2name` are the values of `name2umid` and vice versa. We say that `name2umid` is the **reverse lookup** table (or the **inverse**) for `umid2name`.

What if there are duplicate values? In the word count example, more than one word appears 2 times in the text… How do we deal with that?

Common Pattern: Reverse Lookup and Inversion

```
print(wdcnt)
```

```
{'half': 3, 'a': 3, 'league': 3, 'onward': 1, 'all': 1, 'in': 1, 'the': 2, 'vall
1, 'six': 1, 'hundred': 1}
```

```
wdcnt_reverse = dict()
for w in wdcnt:
    c = wdcnt[w]
    if c in wdcnt_reverse:
        wdcnt_reverse[c].append(w)
    else:
        wdcnt_reverse[c] = [w]
wdcnt_reverse
```

```
{1: ['onward', 'all', 'in', 'valley', 'of', 'death', 'rode', 'six', 'hundred'],
 2: ['the'],
 3: ['half', 'a', 'league']}
```

Here's our original word count dictionary (cropped for readability). Some values (e.g., 1 and 3) appear more than once.

Solution: map values with multiple keys to a list of all keys that had that value.

What if there are duplicate values? In the word count example, more than one word appears 2 times in the text… How do we deal with that?

Common Pattern: Reverse Lookup and Inversion

```
print(wdcnt)
```

```
{'half': 3, 'a': 3, 'league': 3, 'onward': 1, 'all': 1, 'in': 1, 'the': 2, 'vall
1, 'six': 1, 'hundred': 1}
```

```
wdcnt_reverse = dict()
for w in wdcnt:
    c = wdcnt[w]
    if c in wdcnt_reverse:
        wdcnt_reverse[c].append(w)
    else:
        wdcnt_reverse[c] = [w]
wdcnt_reverse
```

```
{1: ['onward', 'all', 'in', 'valley', 'of', 'death', 'rode', 'six', 'hundred'],
 2: ['the'],
 3: ['half', 'a', 'league']}
```

Here's our original word count dictionary (cropped for readability). Some values (e.g., 1 and 3) appear more than once.

Note: there is a more elegant way to do this part of the operation, mentioned in homework 2.

es with multiple keys
at had that value.

What if there are duplicate values? For example, in the word count example, more than one word appears 2 times in the text… How do we deal with that?

Keys Must be Hashable

```
1 d = dict()
2 animals = ['cat','dog','bird','goat']
3 d[animals] = 1.61803
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-77-9fa9089d27b7> in <module>()
      1 d = dict()
      2 animals = ['cat','dog','bird','goat']
----> 3 d[animals] = 1.61803

TypeError: unhashable type: 'list'
```

From the documentation: “All of Python’s immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not.” https://docs.python.org/3/glossary.html#term-hashable

Dictionaries can have dictionaries as values!

Suppose I want to map pairs (x,y) to numbers.

```
1 times_table = dict()
2 for x in range(1,13):
3     if x not in times_table:
4         times_table[x] = dict()
5     for y in range(1,13):
6         times_table[x][y] = x*y
7 times_table[7][9]
```

```
63
```

Each value of `x` maps to another dictionary.

Note: We're putting this if-statement here to illustrate that in practice, we often don't know the order in which we're going to observe the objects we want to add to the dictionary.

Dictionaries can have dictionaries as values!

Suppose I want to map pairs (x,y) to numbers.

```
1 times_table = dict()
2 for x in range(1,13):
3     if x not in times_table:
4         times_table[x] = dict()
5     for y in range(1,13):
6         times_table[x][y] = x*y
7 times_table[7][9]
```

```
63
```

In a few slides we'll see a more natural way to perform this mapping in particular, but this "dictionary of dictionaries" pattern is common enough that it's worth seeing.

Running example: The Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, …

Common pattern: memoization

```
1  def naive_fibo(n):
2      if n < 0:
3          raise ValueError('Negative Fibonacci number?')
4      if n==0:
5          return 0
6      elif n==1:
7          return 1
8      else:
9          return naive_fibo(n-1) + naive_fibo(n-2)
10 for i in range(8,13):
11     print(naive_fibo(i))
```

```
21
34
55
89
144
```

Raise an error. You'll need this in many of your future homeworks.
https://docs.python.org/3/tutorial/errors.html#raising-exceptions

Common pattern: memoization

```
1  def naive_fibo(n):
2      if n < 0:
3          raise ValueError('Negative Fibonacci number?')
4      if n==0:
5          return 0
6      elif n==1:
7          return 1
8      else:
9          return naive_fibo(n-1) + naive_fibo(n-2)
10 for i in range(8,13):
11     print(naive_fibo(i))
```

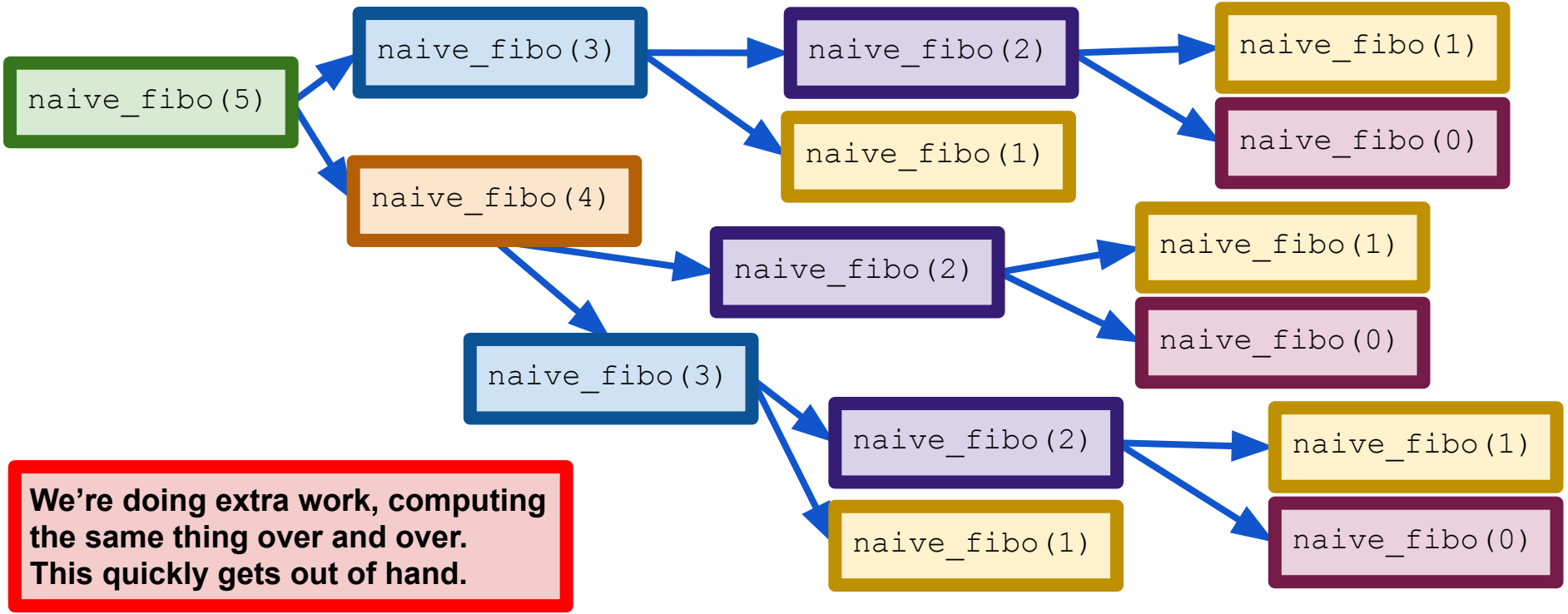
```
21
34
55
89
144
```

Raise an error. You'll need this in many of your future homeworks.
https://docs.python.org/3/tutorial/errors.html#raising-exceptions

This gets slow as soon as the argument gets even moderately big. **Why?**

Common pattern: memoization

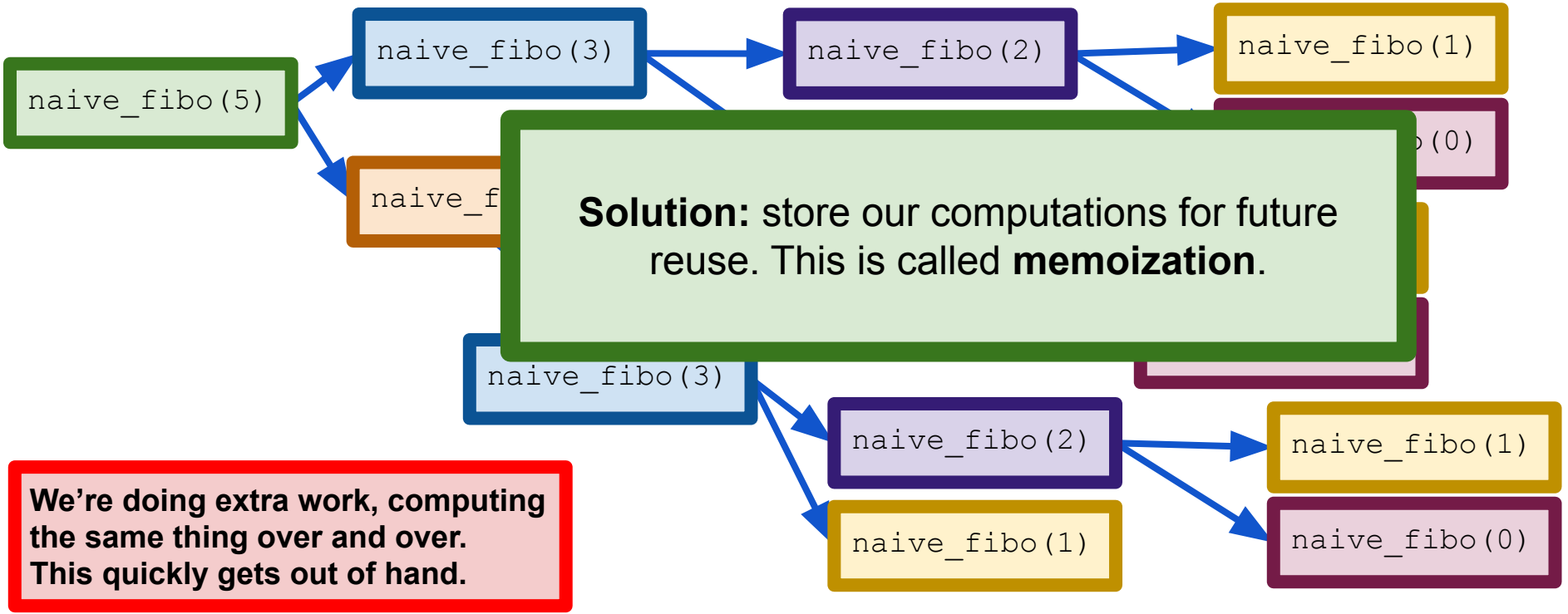
The inefficiency is clear when we draw the **call graph** of the function



We're doing extra work, computing the same thing over and over. This quickly gets out of hand.

Common pattern: memoization

The inefficiency is clear when we draw the **call graph** of the function



Solution: store our computations for future reuse. This is called **memoization**.

We're doing extra work, computing the same thing over and over. This quickly gets out of hand.

Common pattern: memoization

```
1 known = {0:0, 1:1}
2 def fibo(n):
3     if n in known:
4         return known[n]
5     else:
6         f = fibo(n-1) + fibo(n-2)
7         known[n] = f
8         return(f)
9 fibo(30)
```

```
832040
```

This is the dictionary that we'll use for memoization. We'll store `known[n] = fibo(n)` the first time we compute `fibo(n)`, and every time we need it again, we just look it up!

Common pattern: memoization

```
1 known = {0:0, 1:1}
2 def fibo(n):
3     if n in known:
4         return known[n]
5     else:
6         f = fibo(n-1) + fibo(n-2)
7         known[n] = f
8         return(f)
9 fibo(30)
```

```
832040
```

If we already know the n-th Fibonacci number, there's no need to compute it again. Just look it up!

Common pattern: memoization

```
1 known = {0:0, 1:1}
2 def fibo(n):
3     if n in known:
4         return known[n]
5     else:
6         f = fibo(n-1) + fibo(n-2)
7         known[n] = f
8         return(f)
9 fibo(30)
```

```
832040
```

If we don't already know it, we have to compute it, but before we return the result, we memoize it in `known` for future reuse.

Common pattern: memoization

```
1 import time
2 start_time = time.time()
3 naive_fibo(30)
4 time.time() - start_time
```

```
0.8452379703521729
```

```
1 start_time = time.time()
2 fibo(30)
3 time.time() - start_time
```

```
0.00015687942504882812
```

```
1 fibo(100)
```

```
354224848179261915075
```

The time difference is enormous!

Note: this was done with known set to its initial state, so this is a fair comparison.

If you try to do this with `naive_fibo`, you'll be waiting for quite a bit!

```
1 fibo(100)
```

```
354224848179261915075
```

Our memoized Fibonacci function can compute some truly huge numbers!

```
1 fibo(1000)
```

```
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

I cropped this huge number for readability.

```
1 fibo(10000)
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
<ipython-input-116-94cf333eb78c> in <module>()
----> 1 fibo(10000)
```

I cropped some of the error message for readability.

```
----> 6         f = fibo(n-1) + fibo(n-2)
      7         known[n] = f
      8         return(f)

RecursionError: maximum recursion depth exceeded
```

Python runs out of levels of recursion. You can change this maximum recursion depth, but it can introduce instability: https://docs.python.org/3.7/library/sys.html#sys.setrecursionlimit

Common pattern: memoization

```
1 known = {0:0, 1:1}
2 def fibo(n):
3     if n in known:
4         return known[n]
5     else:
6         f = fibo(n-1) + fibo(n-2)
7         known[n] = f
8         return(f)
9 fibo(30)
```

```
832040
```

Congratulations! You've seen your first example of **dynamic programming**! Lots of popular interview questions fall under this purview.
E.g., https://en.wikipedia.org/wiki/Tower_of_Hanoi

Common pattern: memoization

```
1 known = {0:0, 1:1}
2 def fibo(n):
3     if n in known:
4         return known[n]
5     else:
6         f = fibo(n-1) + fibo(n-2)
7         known[n] = f
8         return(f)
9 fibo(30)
```

```
832040
```

Note: the dictionary `known` is declared **outside** the function `fibo`. There is a good reason for this: we don't want `known` to disappear when we finish running `fibo`! We say that `known` is a **global variable**, because it is defined in the "main" program.

Name Spaces: global and local

A **name space** (or **namespace**) is a context in which code is executed

The “outermost” namespace (also called a **frame**) is called `__main__`

- Running from the command line or in Jupyter? You’re in `__main__`
- Often shows up in error messages, something like,
  - “`Error … in __main__: blah blah blah`”
- Variables defined in `__main__` are said to be **global**

Function definitions create their own **local** namespaces

- Variables defined in such a context are called **local**
- Local variables cannot be accessed from outside their frame/namespace

Name Spaces

Example: we have a program simulating a light bulb

Bulb state is represented by a global Boolean variable, `lightbulb_on`

```
1 lightbulb_on = False
2 def lights_on():
3     lightbulb_on = True
4 lights_on()
5 lightbulb_on
```

```
False
```

Bulb is initially off.

Calling this function sets the bulb to the “on” state.

But after calling `lights_on`, the state variable is still `False`. What’s going on?

Name Spaces

```
1 lightbulb_on = False
2 def flip_switch():
3     lightbulb_on = not lightbulb_on
4 flip_switch()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-125-b39d1f83dc2a> in <module>()
      2 def flip_switch():
      3     lightbulb_on = not lightbulb_on
----> 4 flip_switch()

<ipython-input-125-b39d1f83dc2a> in flip_switch()
      1 lightbulb_on = False
      2 def flip_switch():
----> 3     lightbulb_on = not lightbulb_on
      4 flip_switch()

UnboundLocalError: local variable 'lightbulb_on' referenced before assignment
```

The fact that this code causes an error shows what is really at issue. By default, Python treats the variable `lightbulb_on` inside the function definition as being a **different** variable from the `lightbulb_on` defined in the main namespace. This is, generally, a good design. It prevents accidentally changing global state information.

Name Spaces

We have to tell Python that we want `lightbulb_on` to mean the *global* variable

```
1 lightbulb_on = False
2 def flip_switch():
3     global lightbulb_on
4     lightbulb_on = not lightbulb_on
5 flip_switch()
6 lightbulb_on
```

```
True
```

```
1 flip_switch()
2 lightbulb_on
```

```
False
```

Tell Python that we want `lightbulb_on` to refer to the global variable of the same name.

Now, when we call `flip_switch`, the value of `lightbulb_on` is changed successfully.

Warning: this is all well and good, but it is considered best practice to avoid global variables in large programs, as they can make debugging hard. This isn't so crucial for our course, since we won't be building anything especially large, but you should be aware of it.

Important note

Why is this okay, if known isn't declared `global`?

```
1 known = {0:0, 1:1}
2 def fibo(n):
3     if n in known:
4         return known[n]
5     else:
6         f = fibo(n-1) + fibo(n-2)
7         known[n] = f
8         return(f)
9 fibo(30)
```

```
832040
```

`known` is a dictionary, and thus mutable. Maybe mutable variables have special powers and don't have to be declared as global?

Correct answer: global vs local distinction is only important for **variable assignment**. We aren't performing any variable assignment in `fibo`, so no need for the global declaration. Contrast with `lights_on`, where we were reassigning `lightbulb_on`. Variable assignment is **local** by default.

Tuples

Similar to a list, in that it is a sequence of values

But unlike lists, tuples are immutable

Because they are immutable, they are hashable

So we can use tuples where we wanted to key on a list

Documentation:

https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

https://docs.python.org/3/library/stdtypes.html#tuples

Creating Tuples

```
1 t = 1,2,3,4,5
2 t
```

```
(1, 2, 3, 4, 5)
```

```
1 t = (1,2,3,4,5)
2 t
```

```
(1, 2, 3, 4, 5)
```

Tuples created either with “comma notation”, optional parentheses.

Python always displays tuples with parentheses.

```
1 t = 'cat',
2 t
```

```
('cat',)
```

```
1 t = ('cat')
2 t
```

```
'cat'
```

Creating a tuple of one element requires a trailing comma. Failure to include this comma, even with parentheses, yields… not a tuple.

Creating Tuples

```
1 t1 = tuple()
2 t1
```

```
()
```

```
1 t2 = tuple(range(5))
2 t2
```

```
(0, 1, 2, 3, 4)
```

```
1 t3 = tuple('goat')
2 t3
```

```
('g', 'o', 'a', 't')
```

```
1 tuple([[1,2,3],[4,5,6]])
```

```
([1, 2, 3], [4, 5, 6])
```

```
1 print(type(t2))
```

```
<class 'tuple'>
```

Can also create a tuple using the `tuple()` function, which will cast any sequence to a tuple whose elements are those of the sequence.

Tuples are Sequences

```
1 t = ('a', 'b', 'c', 'd', 'e')
2 t[0]
```

```
'a'
```

As sequences, tuples support indexing, slices, etc.

```
1 t[1:4]
```

```
('b', 'c', 'd')
```

```
1 t[-1]
```

```
'e'
```

And of course, sequences have a length.

```
1 len(t)
```

```
5
```

Reminder: sequences support all the operations listed here: https://docs.python.org/3.7/library/stdtypes.html#typesseq

Tuple Comparison

Tuples support comparison, which works analogously to string ordering.

```
1 (1,2,3) < (2,2,3)
```

```
True
```

```
1 (2,2,20) <= (2,2,2)
```

```
False
```

0-th elements are compared. If they are equal, go to the 1-th element, etc.

```
1 (1,2,3) < (1,2,3,4)
```

```
True
```

Just like strings, the “prefix” tuple is ordered first.

```
1 ('cat','dog','goat') > ('dog', 'cat', 'goat')
```

```
False
```

```
1 (1,'cat',(1,2,3)) > (0,'bird',(1,2,0))
```

```
True
```

Tuple comparison is element-wise, so we only need that each element-wise comparison is allowed by Python.

Tuples are Immutable

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

Tuples are immutable, so changing an entry is not permitted.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-c40a1905a6e9> in <module>()
      1 fruits = ('apple', 'banana', 'orange', 'kiwi')
----> 2 fruits[2] = 'grapefruit'

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

As with strings, have to make a new assignment to the variable.

```
1 fruits = fruits[0:2] + ('grapefruit',) + fruits[3:]
2 fruits
```

```
('apple', 'banana', 'grapefruit', 'kiwi')
```

```
1 fruits = fruits[0:2] + 'grapefruit', + fruits[3:]
```

Note: even though `'grapefruit',` is a tuple, Python doesn't know how to parse this line. Use parentheses!

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-f62749483e65> in <module>()
----> 1 fruits = fruits[0:2] + 'grapefruit', + fruits[3:]

TypeError: can only concatenate tuple (not "str") to tuple
```

Useful trick: tuple assignment

```
1 a = 10
2 b = 5
3 print(a, b)
```

```
10 5
```

Tuples in Python allow us to make many variable assignments at once. Useful tricks like this are sometimes called **syntactic sugar**. https://en.wikipedia.org/wiki/Syntactic_sugar

```
tmp = a
a = b
b = tmp
print(a, b)
```

```
10 5
```

Common pattern: swap the values of two variables.

```
1 a = 10
2 b = 5
3 (a,b) = (b,a)
4 print(a, b)
```

```
5 10
```

This line achieves the same end, but in a single assignment statement instead of three, and without the extra variable `tmp`.

Useful trick: tuple assignment

```
1 (x,y,z) = (2*'cat', 0.57721, [1,2,3])
2 (x,y,z)
```

```
('catcat', 0.57721, [1, 2, 3])
```

Tuple assignment requires one variable on the left for each expression on the right.

```
1 (x,y,z) = ('a','b','c','d')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-68-e118c50f83dd> in <module>()
----> 1 (x,y,z) = ('a','b','c','d')

ValueError: too many values to unpack (expected 3)
```

If the number of variables doesn't match the number of expressions, that's an error.

```
1 (x,y,z) = ('a','b')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-69-875f95cea434> in <module>()
----> 1 (x,y,z) = ('a','b')

ValueError: not enough values to unpack (expected 3, got 2)
```

Useful trick: tuple assignment

```
1 email = 'klevin@umich.edu'
2 email.split('@')
```

```
['klevin', 'umich.edu']
```

The `string.split()` method returns a list of strings, obtained by splitting the calling string on the characters in its argument.

```
1 (user,domain) = email.split('@')
2 user
```

```
'klevin'
```

Tuple assignment works so long as the right-hand side is **any** sequence, provided the number of variables matches the number of elements on the right. Here, the right-hand side is a list, `['klevin', 'umich.edu']`.

```
1 domain
```

```
'umich.edu'
```

```
1 (x,y,z) = 'cat'
2 print(x, y, z)
```

```
c a t
```

A string is a sequence, so tuple assignment is allowed. Sequence elements are characters, and indeed, `x`, `y` and `z` are assigned to the three characters in the string.

Tuples as Return Values

```
1 import random
2 def five_numbers(t):
3     t.sort()
4     n = len(t)
5     return (t[0],  t[n//4],  t[n//2], t[(3*n)//4], t[-1])
6 five_numbers([1,2,3,4,5,6,7])
```

```
(1, 2, 4, 6, 7)
```

This function takes a list of numbers and returns a tuple summarizing the list.
https://en.wikipedia.org/wiki/Five-number_summary

```
1 randnumlist = [random.randint(1,100) for x in range(60)]
2 (mini,lowq,med,upq,maxi) = five_numbers(randnumlist)
3 (mini,lowq,med,upq,maxi)
```

```
(3, 27, 54, 73, 98)
```

Test your understanding: what does this list comprehension do?

Tuples as Return Values

More generally, sometimes you want more than one return value

```
1 t = divmod(13,4)
2 t
```

```
(3, 1)
```

```
1 (quotient,remainder) = divmod(13,4)
2 quotient
```

```
3
```

```
1 remainder
```

```
1
```

`divmod` is a Python built-in function that takes a pair of numbers and outputs the quotient and remainder, as a tuple. Additional examples can be found here: https://docs.python.org/3/library/functions.html

Useful trick: variable-length arguments

```
1 def my_min( *args ):
2     return min(args)
3 my_min(1,2,3)
```

```
1
```

```
1 my_min(4,5,6,10)
```

```
4
```

```
1 def print_all( *args ):
2     print(args)
3 print_all('cat', 'dog', 'bird')
```

```
('cat', 'dog', 'bird')
```

```
1 print_all()
```

```
()
```

A parameter name prefaced with * **gathers** all arguments supplied to the function into a tuple.

Note: this is also one of several ways that one can implement optional arguments, though we'll see better ways later in the course.

Gather and Scatter

The opposite of the gather operation is **scatter**

```
1 t = (13, 4)
2 divmod(t)
```

`divmod` takes two arguments, so this is an error.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-106-c7c0a10eef7e> in <module>()
      1 t = (13, 4)
----> 2 divmod(t)

TypeError: divmod expected 2 arguments, got 1
```

```
1 divmod(*t)
```

```
(3, 1)
```

Instead, we have to “untuple” the tuple, using the **scatter** operation. This makes the elements of the tuple into the arguments of the function.

```
1 *t
```

```
  File "<ipython-input-109-f9912a2ca07d>", line 1
    *t
     ^
SyntaxError: can't use starred expression here
```

Note: gather/scatter only works in certain contexts (e.g., for function arguments).

Combining lists: `zip`

Python includes a number of useful functions for combining lists and tuples

```
1 t1 = ['apple', 'orange', 'banana', 'kiwi']
2 t2 = [1, 2, 3, 4]
3 zip(t1,t2)
```

```
<zip at 0x10c95d5c8>
```

`zip()` returns a zip object, which is an **iterator** containing as its elements tuples formed from its arguments.
https://docs.python.org/3/library/functions.html#zip

```
1 for tup in zip(t1,t2):
2     print(tup)
```

```
('apple', 1)
('orange', 2)
('banana', 3)
('kiwi', 4)
```

Iterators are, in essence, objects that support for-loops. All sequences are iterators. Iterators support, crucially, a method `__next__()`, which returns the “next element”. We’ll see this in more detail later in the course.
https://docs.python.org/3/library/stdtypes.html#iterator-types

Combining lists: `zip`

`zip()` returns a zip object, which is an **iterator** containing as its elements tuples formed from its arguments.
https://docs.python.org/3/library/functions.html#zip

```
1 for tup in zip(['a','b','c'],[1,2,3,4]):
2     print(tup)
```

```
('a', 1)
('b', 2)
('c', 3)
```

Given arguments of different lengths, `zip` defaults to the shortest one.

```
1 for tup in zip(['a','b','c','d'],[1,2,3]):
2     print(tup)
```

```
('a', 1)
('b', 2)
('c', 3)
```

`zip` takes any number of arguments, so long as they are all **iterable**. Sequences are iterable.

```
1 for tup in zip([1,2,3],['a','b','c'],'xyz'):
2     print(tup)
```

```
(1, 'a', 'x')
(2, 'b', 'y')
(3, 'c', 'z')
```

Iterables are, essentially, objects that can *become* iterators. We'll see the distinction later in the course.
https://docs.python.org/3/library/stdtypes.html#typeiter

Combining lists: `zip`

```
1 def count_matches(s, t):
2     cnt = 0
3     for (a,b) in zip(s,t):
4         if a==b:
5             cnt += 1
6     return( cnt )
7 count_matches([1,1,2,3,5],[1,2,3,4,5])
```

```
2
```

`zip` is especially useful for iterating over several lists in lockstep.

```
1 count_matches([1,2,3,4,5],[1,2,3])
```

Test your understanding: what should this return?

Combining lists: `zip`

```
1 def count_matches(s, t):
2     cnt = 0
3     for (a,b) in zip(s,t):
4         if a==b:
5             cnt += 1
6     return( cnt )
7 count_matches([1,1,2,3,5],[1,2,3,4,5])
```

```
2
```

`zip` is especially useful for iterating over several lists in lockstep.

```
1 count_matches([1,2,3,4,5],[1,2,3])
```

```
3
```

Test your understanding: what should this return?

Related function: `enumerate()`

```
1 for t in enumerate('goat'):
2     print(t)
```

```
(0, 'g')
(1, 'o')
(2, 'a')
(3, 't')
```

```
1 s = 'goat'
2 for i in range(len(s)):
3     print((i,s[i]))
```

```
(0, 'g')
(1, 'o')
(2, 'a')
(3, 't')
```

`enumerate` returns an **enumerate object**, which is an iterator of (index,element) pairs. It is a more graceful way of performing the pattern below, which we've seen before.
https://docs.python.org/3/library/functions.html#enumerate

Dictionaries revisited

```
1 hist = {'cat':3,'dog':12,'goat':18}
2 hist.items()
```

```
dict_items([('cat', 3), ('dog', 12), ('goat', 18)])
```

```
1 for (k,v) in hist.items():
2     print(k, ':', v)
```

```
cat : 3
dog : 12
goat : 18
```

`dict.items()` returns a dict_items object, an iterator whose elements are (key,value) tuples.

```
1 d = dict([(0,'zero'),(1,'one'),(2,'two')])
2 d
```

```
{0: 'zero', 1: 'one', 2: 'two'}
```

Conversely, we can create a dictionary by supplying a list of (key,value) tuples.

```
1 dict( zip('cat','dog'))
```

```
{'a': 'o', 'c': 'd', 't': 'g'}
```

Tuples as Keys

```
1 name2umid = {('Einstein','Albert'): 'aeinstein',
2  ('Noether','Emmy'): 'enoether',
3  ('Shannon','Claude'): 'cshannon',
4  ('Fan','Ky'): 'kyfan'}
5 name2umid
```

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

In (most) Western countries, the family name is said last (hence “last name”), but it is frequently useful to key on this name before keying on a given name.

```
1 name2umid[('Einstein','Albert')]
```

```
'aeinstein'
```

```
1 sparsemx = dict()
2 sparsemx[(1,4)] = 1
3 sparsemx[(3,5)] = 1
4 sparsemx[(12,13)] = 2
5 sparsemx[(11,13)] = 3
6 sparsemx[(19,13)] = 5
7 sparsemx
```

```
{(1, 4): 1, (3, 5): 1, (11, 13): 3, (12, 13): 2, (19, 13): 5}
```

Keying on tuples is especially useful for representing sparse structures. Consider a 20-by-20 matrix in which most entries are zeros. Storing all the entries requires 400 numbers, but if we only record the entries that are nonzero...

Data Structures: Lists vs Tuples

Use a **list** when:

- Length is not known ahead of time and/or may change during execution
- Frequent updates are likely

Use a **tuple** when:

- The set is unlikely to change during execution
- Need to key on the set (i.e., require immutability)
- Want to perform multiple assignment or for use in variable-length arg list

Most code you see will use lists, because mutability is quite useful

Intermission

Week 3 Practice Problems

On Canvas, in Files/in-class practice/week3practice.ipynb

Name Spaces -- revisited

In [2]:
```
lightbulb_on = False

def flip_switch():
    lightbulb_on = not lightbulb_on

flip_switch()
lightbulb_on
```

```
UnboundLocalError: local variable 'lightbulb_on' referenced before assignment
```

In [4]:
```
lightbulb_on = False

def print_wrong_lightbulb_status():
    print(not lightbulb_on)

print_wrong_lightbulb_status()
```

```
True
```

In [6]:
```
lightbulb_on = False

def turn_on_wrong_light():
    lightbulb_on = True

turn_on_wrong_light()
lightbulb_on
```

Out[6]:
```
False
```

Name Spaces -- revisited

We have to tell Python that we want `lightbulb_on` to mean the *global* variable

```
1 lightbulb_on = False
2 def flip_switch():
3     global lightbulb_on
4     lightbulb_on = not lightbulb_on
5 flip_switch()
6 lightbulb_on
```

```
True
```

```
1 flip_switch()
2 lightbulb_on
```

```
False
```

Tell Python that we want `lightbulb_on` to refer to the global variable of the same name.

Now, when we call `flip_switch`, the value of `lightbulb_on` is changed successfully.

Warning: this is all well and good, but it is considered best practice to avoid global variables in large programs, as they can make debugging hard. This isn't so crucial for our course, since we won't be building anything especially large, but you should be aware of it.

Important note -- revisited

Why is this okay, if known isn't declared `global`?

```
1 known = {0:0, 1:1}
2 def fibo(n):
3     if n in known:
4         return known[n]
5     else:
6         f = fibo(n-1) + fibo(n-2)
7         known[n] = f
8         return(f)
9 fibo(30)
```

```
832040
```

`known` is a dictionary, and thus mutable. Maybe mutable variables have special powers and don't have to be declared as global?

Correct answer: global vs local distinction is only important for **variable assignment**. We aren't performing any variable assignment in `fibo`, so no need for the global declaration. Contrast with `lights_on`, where we were reassigning `lightbulb_on`. Variable assignment is **local** by default.

Persistent data

So far, we only know how to write “transient” programs

- Data disappears once the program stops running

Files allow for **persistence**

- Work done by a program can be saved to disk...
- ...and picked up again later for other uses.

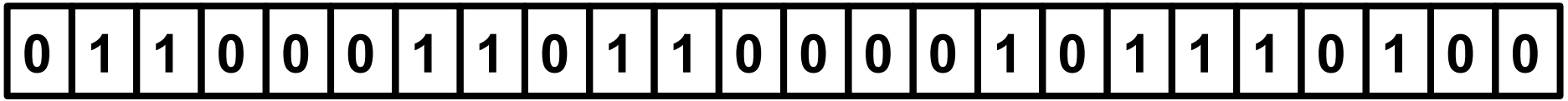
Examples of persistent programs:

- Operating systems
- Databases
- Servers

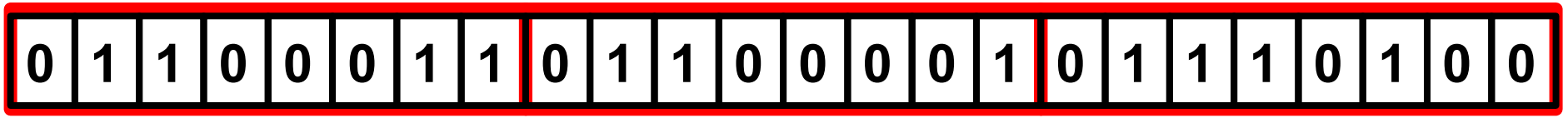
Key idea: Program information is stored permanently (e.g., on a hard drive), so that we can start and stop programs without losing **state** of the program (values of variables, where we are in execution, etc).

Reading and Writing Files

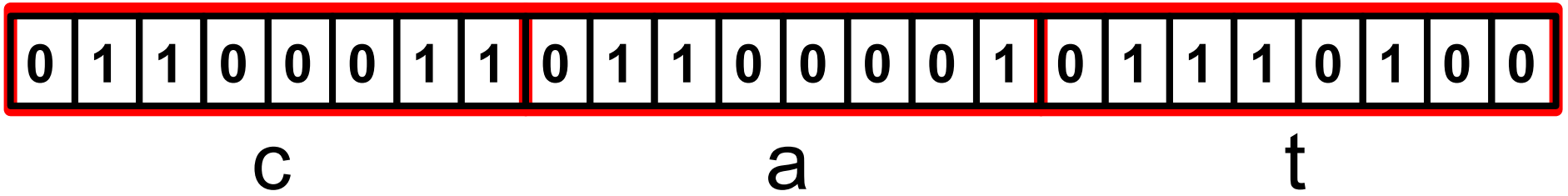
Underlyingly, every file on your computer is just a string of bits…



...which are broken up into (for example) bytes…



...which correspond (in the case of text) to characters.



Reading files

This is the command line. We'll see lots more about this later, but for now, it suffices to know that the command `cat` prints the contents of a file to the screen.

```
$ cat demo.txt
This is a demo file.
It is a text file, containing three lines of text.
Here is the third line.
$
```

```
1 f = open('demo.txt')
2 type(f)
```

```
_io.TextIOWrapper
```

Open the file `demo.txt`. This creates a **file object** `f`.
https://docs.python.org/3/glossary.html#term-file-object

```
1 f.readline()
```

```
'This is a demo file.\n'
```

Provides a method for reading a single line from the file. The string `'\n'` is a **special character** that represents a new line. More on this soon.

Reading files

```
$ cat demo.txt
This is a demo file.
It is a text file, containing three lines of text.
Here is the third line.
$
```

```
1 f = open('demo.txt')
2 f.readline()
```

```
'This is a demo file.\n'
```

```
1 f.readline()
```

```
'It is a text file, containing three lines of text.\n'
```

```
1 f.readline()
```

```
'Here is the third line.\n'
```

```
1 f.readline()
```

```
''
```

Each time we call `f.readline()`, we get the next line of the file...

...until there are no more lines to read, at which point the `readline()` method returns the empty string whenever it is called.

Reading files

```
1 f = open('demo.txt')
2 for line in f:
3     for wd in line.split():
4         print(wd.strip('.,'))
```

```
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

We can treat `f` as an iterator, in which each iteration gives us a line of the file.

Iterate over each word in the line (splitting on `' '` by default).

Remove the trailing punctuation from the words of the file.

`open()` provides a bunch more (optional) arguments, some of which we'll discuss later.
https://docs.python.org/3/library/functions.html#open

Reading files

```
1 with open('demo.txt') as f:
2     for line in f:
3         for wd in line.split():
4             print(wd.strip('.,'))
```

```
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

You may often see code written this way, using the `with` keyword. We'll see it in detail later. For now, it suffices to know that this is equivalent to what we did on the previous slide.

From the documentation: "It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point." https://docs.python.org/3/reference/compound_stmts.html#with

In plain English: the `with` keyword does a bunch of error checking and cleanup for you, automatically.

Writing files

Open the file in **write** mode. If the file already exists, this creates it anew, deleting its old contents.

```
1 f = open('animals.txt', 'w')
2 f.read()
```

If I try to read a file in write mode, I get an error.

```
---------------------------------------------------------------------------
UnsupportedOperation                                 Traceback (most recent call last)
<ipython-input-29-3b1ef477003a> in <module>()
      1 f = open('animals.txt', 'w')
----> 2 f.read()

UnsupportedOperation: not readable
```

Write to the file. This method returns the number of characters written to the file. Note that `'\n'` counts as a single character, the new line.

```
1 f.write('cat\n')
2 f.write('dog\n')
3 f.write('bird\n')
4 f.write('goat\n')
```

```
5
```

Writing files

```
1 f = open('animals.txt', 'w')
2 f.write('cat\n')
3 f.write('dog\n')
4 f.write('bird\n')
5 f.write('goat\n')
6 f.close()
```

Open the file in **write** mode.
This overwrites the version of the file created in the previous slide.

Each write appends to the end of the file.

When we're done, we close the file. This happens automatically when the program ends, but its good practice to close the file as soon as you're done.

```
1 f = open('animals.txt', 'r')
2 for line in f:
3     print(line, end="")
```

```
cat
dog
bird
goat
```

Now, when I open the file for reading, I can print out the lines one by one.

The lines of the file already include newlines on the ends, so override Python's default behavior of printing a newline after each line.

Aside: Formatting Strings

```
1 x = 23
2 print('x = %d' % x)
```

```
x = 23
```

```
1 animal = 'unicorn'
2 print('My pet %s' % animal)
```

```
My pet unicorn
```

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y,x/y))
```

```
2.718000 divided by 1.618000 is 1.679852
```

```
1 print('%.3f divided by %.3f is %.8f' % (x,y,x/y))
```

```
2.718 divided by 1.618 is 1.67985167
```

Python provides tools for formatting strings. Example: easier way to print an integer as a string.

`%d` : integer
`%s` : string
`%f` : floating point
More information:
https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

Can further control details of formatting, such as number of significant figures in printing floats.

Newer features for similar functionality:
https://docs.python.org/3/reference/lexical_analysis.html#f-strings
https://docs.python.org/3/library/stdtypes.html#str.format

Aside: Formatting Strings

Note: Number of formatting arguments must match the length of the supplied tuple!

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y,x/y,1.0))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-eb736fce3612> in <module>()
      1 x = 2.718; y = 1.618
----> 2 print('%f divided by %f is %f' % (x,y,x/y,1.0))

TypeError: not all arguments converted during string formatting
```

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-b2e6a26d3415> in <module>()
      1 x = 2.718; y = 1.618
----> 2 print('%f divided by %f is %f' % (x,y))

TypeError: not enough arguments for format string
```

Saving objects to files: `pickle`

Sometimes it is useful to be able to turn an object into a string

```
1 import pickle
2 t1 = [1,'two',3.0]
3 s = pickle.dumps(t1)
4 s
```

`pickle.dumps()` (short for “dump string”) creates a **binary string** representing an object.

```
b'\x80\x03]q\x00(K\x01X\x03\x00\x00\x00twoq\x01G@\x08\x00\x00\x00\x00\x00\x00e.'
```

This is a raw binary string that encodes the list `t1`. Each symbol encodes one byte. More detail later in the course.
https://docs.python.org/3.7/library/functions.html#func-bytes
https://en.wikipedia.org/wiki/ASCII

```
1 t2 = pickle.loads(s)
2 t1==t2
```

```
True
```

```
1 t1 is t2
```

```
False
```

Saving objects to files: `pickle`

Sometimes it is useful to be able to turn an object into a string

```
1 import pickle
2 t1 = [1,'two',3.0]
3 s = pickle.dumps(t1)
4 s
```

```
b'\x80\x03]q\x00(K\x01X\x03\x00\x00\x00twoq\x01G@\x08\x00\x00\x00\x00\x00\x00e.'
```

We can now use this string to store (a representation of) the list referenced by `t1`. We can write it to a file for later reuse, use it as a key in a dictionary, etc.

```
1 t2 = pickle.loads(s)
2 t1==t2
```

```
True
```

Later on, to “unpickle” the string and turn it back into an object, we use `pickle.loads()` (short for “load string”).

```
1 t1 is t2
```

```
False
```

Important point: pickling stores a representation of the value, not the variable! So after this assignment, `t1` and `t2` are equivalent...

...but not identical.

Locating files: the `os` module

```
1 import os
2 cwd = os.getcwd()
3 cwd
```

```
'/Users/keith/demo/L6_Files'
```

`os` module lets us interact with the operating system.
https://docs.python.org/3.7/library/os.html

`os.getcwd()` returns a string corresponding to the **current working directory**.

```
1 os.listdir()
```

```
['data', 'scripts']
```

```
1 os.listdir('data')
```

```
['numbers.txt', 'pi.txt']
```

`os.listdir()` lists the contents of its argument, or the current directory if no argument.

```
1 os.chdir('data')
2 os.getcwd()
```

```
'/Users/keith/demo/L6_Files/data'
```

`os.chdir()` changes the working directory. After calling `chdir()`, we're in a different cwd.

Locating files: the `os` module

```
1 import os
2 cwd = os.getcwd()
3 cwd
```

```
'/Users/keith/demo/L6_Files'
```

This is called a **path**. It starts at the **root directory**, `'/'`, and describes a sequence of nested directories.

```
1 os.listdir()
```

```
['data', 'scripts']
```

```
1 os.listdir('data')
```

```
['numbers.txt', 'pi.txt']
```

A path from the root to a file or directory is called an **absolute path**. A path from the current directory is called a **relative path**.

```
1 os.path.abspath('data/pi.txt')
```

```
'/Users/keith/demo/L6_Files/data/pi.txt'
```

Use `os.path.abspath` to get the absolute path to a file or directory.

Locating files: the `os` module

```
1 import os
2 os.chdir('/Users/keith/demo/L6_Files')
3 os.listdir('data')
```

```
['extra', 'numbers.txt', 'pi.txt']
```

```
1 os.path.exists('data/pi.txt')
```

```
True
```

```
1 os.path.exists('data/nonsense.txt')
```

```
False
```

Check whether or not a file/directory exists.

```
1 os.path.isdir('data/extra')
```

```
True
```

```
1 os.path.isdir('data/numbers.txt')
```

```
False
```

Check whether or not this is a directory. `os.path.isfile()` works analogously.

Handling errors: try/catch statements

Sometimes when an error occurs, we want to try and recover

Rather than just giving up and having Python yell at us.

Python has a special syntax for this: `try:... except:...`

Basic idea: try to do something, and if an error occurs, try something else.

Example: try to open a file for reading.

If that fails (e.g., because the file doesn't exist) look for the file elsewhere

Handling errors: try/catch statements

```
1 import os
2 os.listdir()
```

```
['backup_file.txt', 'data', 'scripts']
```

```
1 try:
2     f = open('nonsense.txt')
3 except:
4     f = open('backup_file.txt')
5 f.read()
```

```
'This is a backup file.\n'
```

Python attempts to execute the code in the `try` block. If that runs successfully, then we continue on.

If the `try` block fails (i.e., if there's an **exception**), then we run the code in the `except` block.

Programmers call this kind of construction a **try/catch statement**, even though the Python syntax uses `try/except` instead.

Handling errors: try/catch statements

```
1 import os
2 os.listdir()
```

```
['backup_file.txt',
```

Note: this pattern is really only necessary in particular situations where you know how you want to recover from the error. Otherwise, it's better to just raise an error. I show it here because you'll see this pattern frequently "in the wild".

```
1 try:
2     f = open('nonsense.txt')
3 except:
4     f = open('backup_file.txt')
5 f.read()
```

```
'This is a backup file.\n'
```

cute the code in
ns successfully,
then we continue on.

If the `try` block fails (i.e., if there's an **exception**), then we run the code in the `except` block.

Programmers call this kind of construction a **try/catch statement**, even though the Python syntax uses `try/except` instead.

Raising exceptions

In [17]:
```
1 def my_sum(a, b):
2     if not isinstance(a, (int, float, complex)):
3         raise TypeError("%s is not a number" % a)
4     if not isinstance(b, (int, float, complex)):
5         raise TypeError("%s is not a number" % b)
6     return a + b
```

In [18]:
```
1 my_sum(4, 6)
```

Out[18]:
```
10
```

In [19]:
```
1 my_sum(4, "hamburger")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-19-fdb8cf45d9c1> in <module>
----> 1 my_sum(4, "hamburger")

<ipython-input-17-57303bdc0e56> in my_sum(a, b)
      3         raise TypeError("%s is not a number" % a)
      4     if not isinstance(b, (int, float, complex)):
----> 5         raise TypeError("%s is not a number" % b)
      6     return a + b

TypeError: hamburger is not a number
```

Catching Exceptions

In [27]:
```
def my_sum(a, b):
    if not isinstance(a, (int, float, complex)):
        raise TypeError("%s is not a number" % a)
    if not isinstance(b, (int, float, complex)):
        raise TypeError("%s is not a number" % b)
    return a + b
```

In [28]:
```
def sum_or_cheeseburger(a, b):
    try:
        the_sum = my_sum(a, b)
        return the_sum
    except TypeError:
        return "cheeseburger"
```

In [29]:
```
sum_or_cheeseburger(4.5, 33)
```

Out[29]: 37.5

In [30]:
```
sum_or_cheeseburger(None, 1e-9)
```

Out[30]: 'cheeseburger'

Writing modules

Python provides modules (e.g., `math`, `os`, `time`)

But we can also write our own, and import from them with same syntax

```
1 import prime
2 prime.is_prime(2)
```

```
True
```

```
1 prime.is_prime(3)
```

```
True
```

```
1 prime.is_prime(1)
```

```
False
```

```
1 prime.is_prime(23)
```

```
True
```

prime.py

```
import math

def is_prime(n):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(n))
        for k in range(2,ulim+1):
            if n%k==0:
                return False
        return True
```

Writing modules

Import everything defined in `prime`, so we can call it without the prefix. Can also import specific functions: `from prime import is_square`

```
1 from prime import *
2 is_prime(7)
```

```
True
```

```
1 is_square(7)
```

```
False
```

```
1 is_prime(373)
```

```
True
```

Caution: be careful that you don't cause a collision with an existing function or a function in another module!

prime.py

```
1  import math
2
3  def is_prime(n):
4      if n <= 1:
5          return False
6      elif n==2:
7          return True
8      else:
9          ulim = math.ceil(math.sqrt(n))
10         for k in range(2,ulim+1):
11             if n%k==0:
12                 return False
13         return True
14 def is_square(n):
15     r = int(math.sqrt(n))
16     return(r*r==n or (r+1)*(r+1)==n)
```