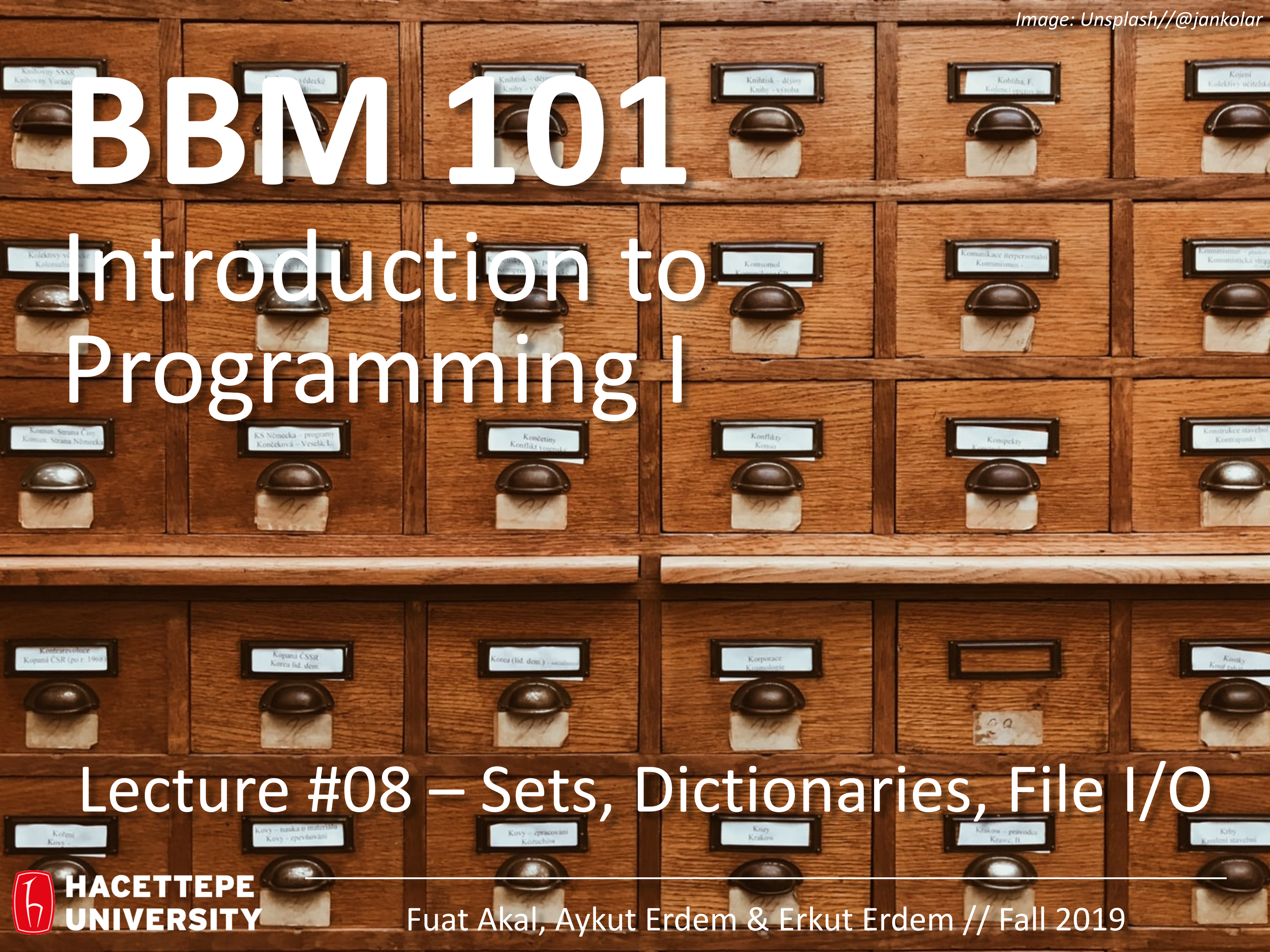


Image: Unsplash//@jankolar

BBM 101

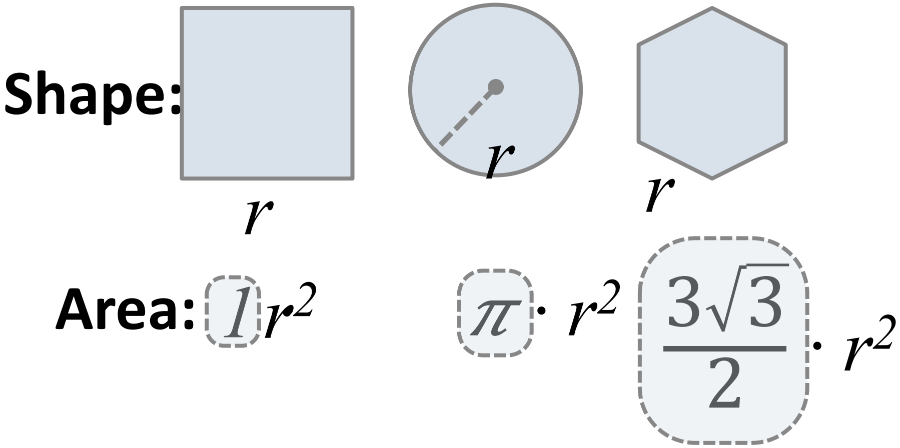
Introduction to Programming I

Lecture #08 – Sets, Dictionaries, File I/O

HACETTEPE UNIVERSITY

Fuat Akal, Aykut Erdem & Erkut Erdem // Fall 2019

Last time... **Higher-Order Functions**



Finding common structure allows for shared implementation!

The built-in function `filter(f, seq)`

```
primes = filter(is_prime, range(11))
```

The built-in function `map(f, seq)`

```
squares = map(lambda x : x ** 2, range(11))
```

Function currying

```
def make_adder(n):
    return lambda k: n + k
```

```
square = lambda x: x * x
```

VS

```
def square(x):
    return x * x
```

Lecture Overview

- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries
- File I/O

Disclaimer: Much of the material and slides for this lecture were borrowed from
—Ruth Anderson, Michael Ernst and Bill Howe's University of Washington CSE 140 class,
—Ana Bell, Eric Grimson, John Guttag's MIT 6.0001 class
—Keith Levin's University of Michigan STATS 507 class

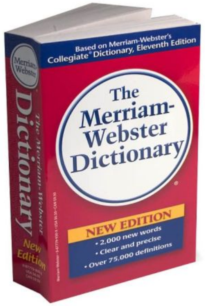
Recall: Data Structures

- A *data structure* is way of organizing data
  - Each data structure makes certain operations convenient or efficient
  - Each data structure makes certain operations inconvenient or inefficient

Recall: Collections

- List: ordered
- Tuple: unmodifiable list
- **Set**: unordered, no duplicates
- **Dictionary**: maps from values to values

  Example: word → definition


Based on Merriam-Webster's
Collegiate Dictionary, Eleventh Edition
The Merriam-Webster Dictionary
NEW EDITION
• 2,000 new words
• Clear and precise
• Over 75,000 definitions
Merriam-Webster
Dictionary
The Merriam-Webster
New Edition

Lecture Overview

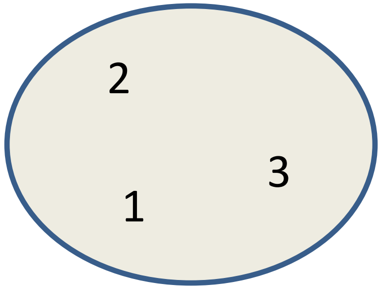
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries
- File I/O

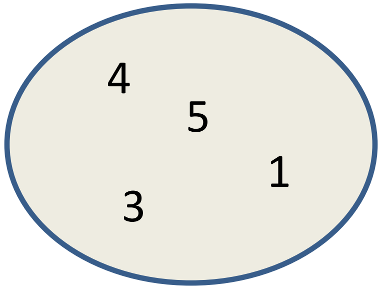
Sets

- Mathematical set: a collection of values, without duplicates or order

- Order does not matter
  { 1, 2, 3 } == { 3, 2, 1 }

- No duplicates
  { 3, 1, 4, 1, 5 } == { 5, 4, 3, 1 }





- For every data structure, ask:
  - How to create
  - How to query (look up) and perform other operations
    - (Can result in a new set, or in some other datatype)
  - How to modify

  Answer: http://docs.python.org/3/library/stdtypes.html#set

Creating a Set

- Construct from a **list:**

```
odd = set([1, 3, 5])
prime = set([2, 3, 5])
empty = set([])
```

Set Operations

```
odd = set([ 1, 3, 5 ])
prime = set([ 2, 3, 5 ])
```

membership $\in$	Python: `in`	`4 in prime` $\Rightarrow$ False
union $\cup$	Python: `|`	`odd | prime` $\Rightarrow$ { 1, 2, 3, 5 }
intersection $\cap$	Python: `&`	`odd & prime` $\Rightarrow$ { 3, 5 }
difference \ or -	Python: `–`	`odd – prime` $\Rightarrow$ { 1 }

Think in terms of **set operations**,
not in terms of iteration and element operations

– Shorter, clearer, less error-prone, faster

Although we can do iteration over sets:

```
    # iterates over items in arbitrary order
    for item in myset:
      …
```

But we *cannot* index into a set to access a specific element.

Modifying a Set

- **Add** one element to a set:

```
myset.add(newelt)
myset = myset | set([newelt])
```

- **Remove** one element from a set:

```
myset.remove(elt) # elt must be in myset or raises err
myset.discard(elt)# never errs
```

What would this do?

```
myset = myset – set([newelt])
```

- Choose and remove some element from a set:

```
myset.pop()
```

Practice with Sets

```
z = set([5,6,7,8])
y = set([1,2,3,"foo",1,5])
k = z & y
j = z | y
m = y – z
z.add(9)
```

```
z: {8, 9, 5, 6, 7}
y: {1, 2, 3, 5, 'foo'}
k: {5}
j: {1, 2, 3, 5, 6, 7, 8, 'foo'}
m: {1, 2, 3, 'foo'}
```

List vs. Set Operations (1)

Find the common elements **in both** list1 and list2:

```
out1 = []
for i in list2:
    if i in list1:
        out1.append(i)
```

or

```
out1 = [i for i in list2 if i in list1]
```

Find the common elements in both set1 and set2:

```
set1 & set2
```

Much shorter, clearer, easier to write!

List vs. Set Operations (2)

Find the elements in **either** list1 or list2 (**or both**) (without duplicates):

```
out2 = list(list1)      # make a copy
for i in list2:
    if i not in list1:  # don’t append elements
   out2.append(i)       # already in out2
```

or

```
out2 = list1+list2
for i in out1:       # out1 (from previous example),
   out2.remove(i)    # common elements in both lists
                     # Remove common elements
```

Find the elements in either set1 or set2 (or both):

```
set1 | set2
```

List vs. Set Operations (3)

Find the elements in **either list but not in both**:

```
out3 = []
for i in list1+list2:
    if i not in list1 or i not in list2:
        out3.append(i)
```

Find the elements in either set but not in both:

```
set1 ^ set2     # symmetric difference
```

Set Elements

- Set elements must be immutable values
  - int, float, bool, string, *tuple*
  - *not*: list, set, dictionary
- Goal: only set operations change the set
  - after “`myset.add(x)`”, `x in myset` ⇒ True
  - `y in myset` always evaluates to the same value

  Both conditions should hold until `myset` itself is changed

Set Elements

- Mutable elements can violate these goals

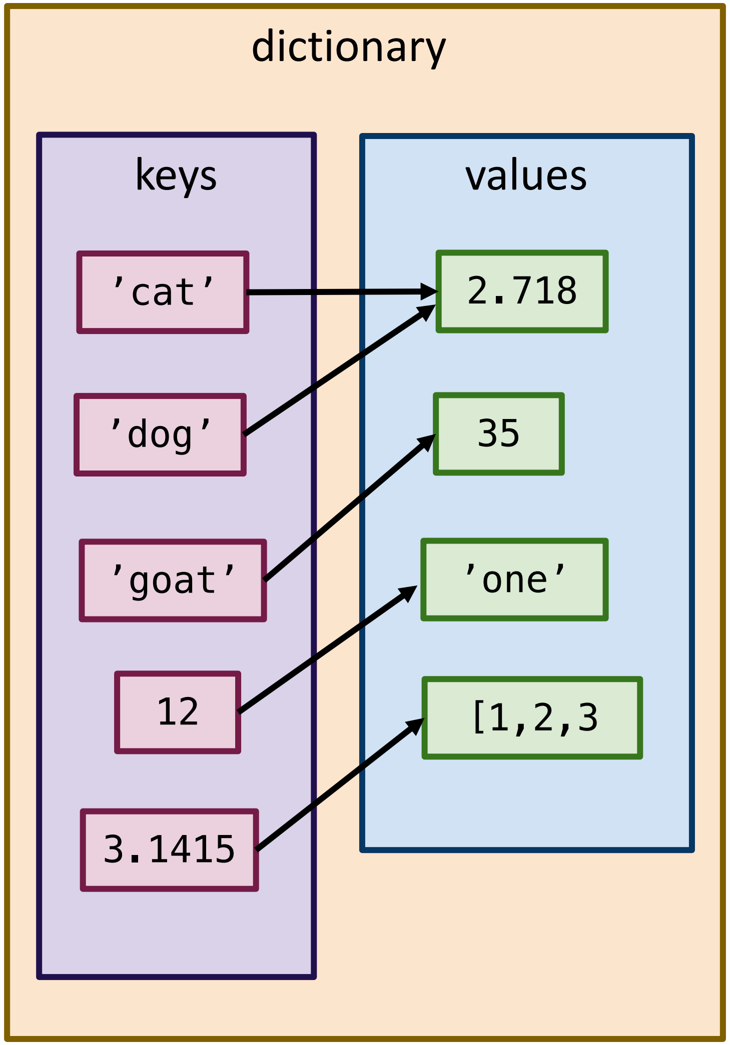
```
list1 = ["a", "b"]
list2 = list1
list3 = ["a", "b"]
myset = { list1 }                ⇐ Hypothetical; actually illegal in Python
                                     TypeError: unhashable type: 'list'

list1 in myset                   ⇒ True
list3 in myset                   ⇒ True
list2.append("c")                ⇐ modifying myset  “indirectly” would
                                   lead to different results

list1 in myset                    ⇒ ???
list3 in myset                    ⇒ ???
```

Lecture Overview

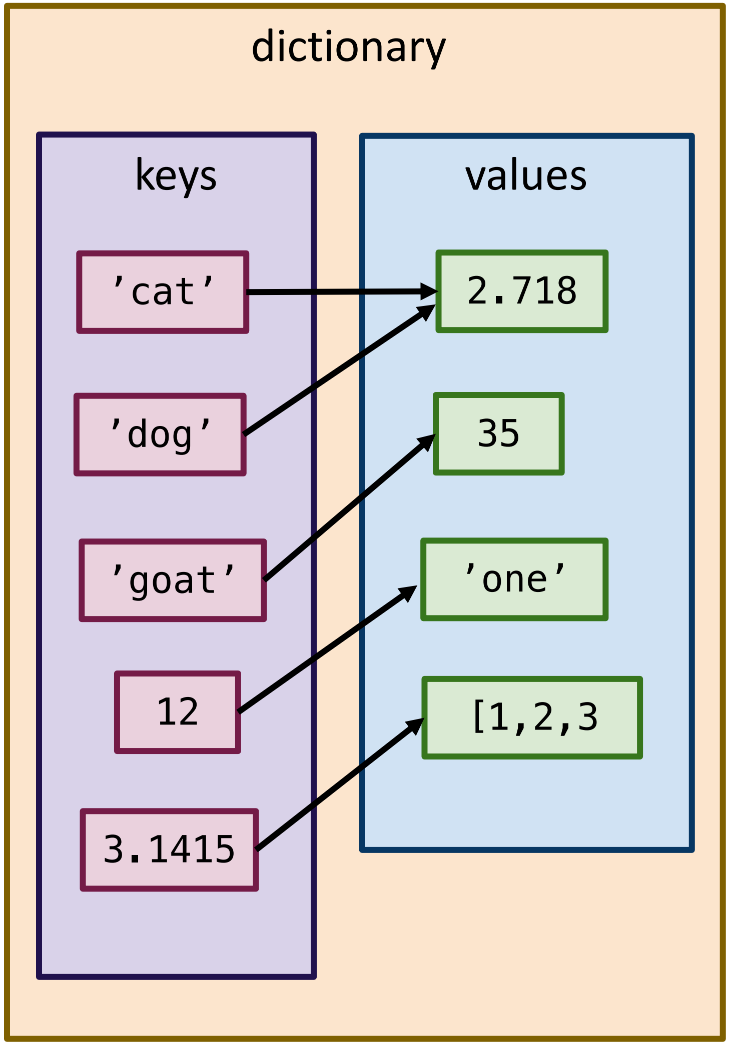
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries
- File I/O

Dictionaries

- 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**)
- Each key associated with one (and only one) value **key-value pairs**, sometimes called **items**
- Like a function f: keys -> values



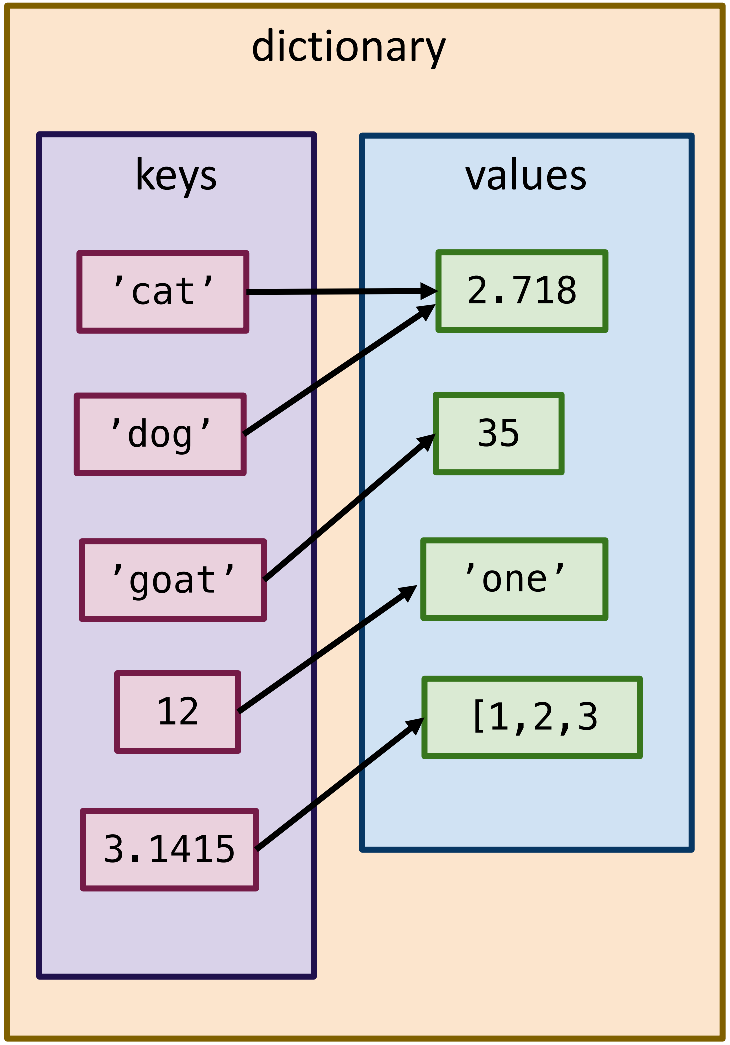
Dictionaries



- Dictionary maps keys to values.
- E.g., 'cat' mapped to the float 2.718
- 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 HU-CENG unique names to people

Accessing a Dictionary



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

```
>> example_dict['goat']
35

>> example_dict['cat']
2.718

>> example_dict['dog']
2.718

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

>> example_dict[12]
'one'
```

Accessing a Dictionary

<u>Example:</u>
Hacettepe University IT wants to store the correspondence btw the usernames (HU-CENG IDs) of students to their actual names. A dictionary is a very natural data structure for this.

```
>>> huceng2name = dict()
>>> huceng2name[‘aeinstein’] = ‘Albert Einstein’
>>> huceng2name[‘kyfan’] = ‘Ky Fan’
>>> huceng2name[‘enoether’] = ‘Emmy Noether’
>>> huceng2name[‘cshannon’] = ‘Claude Shannon’

>>> huceng2name[‘cshannon’]
’Claude Shannon’

>>> huceng2name[‘enoether’]
‘Emmy Noether’

>>> huceng2name[‘enoether’] = ‘Amalie Emmy Noether’
>>> huceng2name[‘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.

```
>>> huceng2name = dict()
>>> huceng2name[‘aeinstein’] = ‘Albert Einstein’
>>> huceng2name[‘kyfan’] = ‘Ky Fan’
>>> huceng2name[‘enoether’] = ‘Emmy Noether’
>>> huceng2name[‘cshannon’] = ‘Claude Shannon’

>>> huceng2name[‘cshannon’]
’Claude Shannon’

>>> huceng2name[‘enoether’]
‘Emmy Noether’

>>> huceng2name[‘enoether’] = ‘Amalie Emmy Noether’
>>> huceng2name[‘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.

```
>>> huceng2name = dict()
>>> huceng2name[‘aeinstein’] = ‘Albert Einstein’
>>> huceng2name[‘kyfan’] = ‘Ky Fan’
>>> huceng2name[‘enoether’] = ‘Emmy Noether’
>>> huceng2name[‘cshannon’] = ‘Claude Shannon’

>>> huceng2name[‘cshannon’]
’Claude Shannon’

>>> huceng2name[‘enoether’]
‘Emmy Noether’

>>> huceng2name[‘enoether’] = ‘Amalie Emmy Noether’
>>> huceng2name[‘enoether’]
‘Amalie Emmy Noether’
```

Creating and populating a dictionary

```
>>> huceng2name = dict()
>>> huceng2name[‘aeinstein’] = ‘Albert Einstein’
>>> huceng2name[‘kyfan’] = ‘Ky Fan’
>>> huceng2name[‘enoether’] = ‘Emmy Noether’
>>> huceng2name[‘cshannon’] = ‘Claude Shannon’
```

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

```
>>> huceng2name[‘cshannon’]
’Claude Shannon’

>>> huceng2name[‘enoether’]
‘Emmy Noether’
```

```
>>> huceng2name[‘enoether’] = ‘Amalie Emmy Noether’
>>> huceng2name[‘enoether’]
‘Amalie Emmy Noether’
```

Creating and populating a dictionary

```
>>> huceng2name = dict()
>>> huceng2name[‘aeinstein’] = ‘Albert Einstein’
>>> huceng2name[‘kyfan’] = ‘Ky Fan’
>>> huceng2name[‘enoether’] = ‘Emmy Noether’
>>> huceng2name[‘cshannon’] = ‘Claude Shannon’

>>> huceng2name[‘cshannon’]
’Claude Shannon’

>>> huceng2name[‘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.

```
>>> huceng2name[‘enoether’] = ‘Amalie Emmy Noether’
>>> huceng2name[‘enoether’]
‘Amalie Emmy Noether’
```

Displaying Items

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

```
>>> example_dic
{3.1415: [1, 2, 3], 12: ‘one’, ‘cat’: 2.718, ‘dog’: 2.718, ‘goat’: 35}

>>> huceng2name
{‘aeinstein’: ‘Albert Einstein’,
 ‘cshannon’: ‘Claude Shannon’,
 ‘enoether’: ‘Amalie Emmy Noether’,
 ‘kyfan’: ‘Ky Fan’}
```

… we can also use that format to create a new dictionary.

```
>>> huceng2name = {‘aeinstein’: ‘Albert Einstein’,
      ‘cshannon’: ‘Claude Shannon’,
      ‘enoether’: ‘Amalie Emmy Noether’,
      ‘kyfan’: ‘Ky Fan’}
>>> huceng2name[‘kyfan’]
‘Ky Fan’
```

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

Dictionaries have a length

```
>>> huceng2name
{‘aeinstein’: ‘Albert Einstein’,
 ‘cshannon’: ‘Claude Shannon’,
 ‘enoether’: ‘Amalie Emmy Noether’,
 ‘kyfan’: ‘Ky Fan’}
```

Length of a dictionary is just the number of items.

```
>>> len(huceng2name)
4
```

Empty dictionary has length 0.

```
>>> d = dict()
>>> len(d)
0
```

Checking set membership

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

```
>>> huceng2name
{‘aeinstein’: ‘Albert Einstein’,
 ‘cshannon’: ‘Claude Shannon’,
 ‘enoether’: ‘Amalie Emmy Noether’,
 ‘kyfan’: ‘Ky Fan’}

>>> ‘akolmogorov’ in huceng2name
False

>>> ‘enoether’ in huceng2name
True
```

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

```
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
```

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

```
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
```

The random module supports a bunch of random number generation operations. 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

```
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.

```
>>> 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

```
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

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

```
>>> 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()
0.000219676971435547
```

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

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

```
>>> import time
>>> start_time = time.time()
>>> 8675309 in list_of_numbers
>>> time.time() – start_time()
0.10922789573669434
```

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()
0.000219676971435547
```

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

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

```
>>> 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()
0.00022196769714355 47
```

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

Checking set membership: Fast and Slow

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

```
>>> 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
```

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.

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

Traversing a dictionary

- Suppose we have a dictionary representing word counts...
- ...and now we want to display the counts for each word.

```
>>> for w in wdcnt:
        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.

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

```
>>> huceng2name
{‘aeinstein’: ‘Albert Einstein’,
 ‘cshannon’: ‘Claude Shannon’,
 ‘enoether’: ‘Amalie Emmy Noether’,
 ‘kyfan’: ‘Ky Fan’}

>>> name2huceng = dict()
    for uname in huceng2name:
        truename = huceng2name[uname]
        name2huceng[truename] = uname

>>> name2huceng
{‘Albert Einstein’: ‘aeinstein’,
 ‘Amalie Emmy Noether’: ‘enoether’,
 ‘Claude Shannon’: ‘cshannon’,
 ‘Ky Fan’: ‘kyfan’}
```

The keys of huceng2name are the values of name2huceng and vice versa. We say that name2huceng is the **reverse lookup** table (or the **inverse**) for huceng2name.

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

```
>>> huceng2name
{‘aeinstein’: ‘Albert Einstein’,
 ‘cshannon’: ‘Claude Shannon’,
 ‘enoether’: ‘Amalie Emmy Noether’,
 ‘kyfan’: ‘Ky Fan’}

>>> name2huceng = dict()
    for uname in huceng2name:
        truename = huceng2name[uname]
        name2huceng[truename] = uname

>>> name2huceng
{‘Albert Einstein’: ‘aeinstein’,
 ‘Amalie Emmy Noether’: ‘enoether’,
 ‘Claude Shannon’: ‘cshannon’,
 ‘Ky Fan’: ‘kyfan’}
```

The keys of huceng2name are the values of name2huceng and vice versa. We say that name2huceng is the **reverse lookup** table (or the **inverse**) for huceng2name.

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?

Keys must be hashable!

```
>>> d = dict()
>>> animals = [‘cat’, ‘dog’, ‘bird’, ‘goat’]
>>> d[animals] = 1.61803

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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 we want to map pairs (x,y) to numbers.

```
>>> times_table = dict()

>>> for x in range(1,13):
        if x not in times_table:
            times_table[x] = dict()
        for y in range(1,13):
            times_table[x][y] = x*y

>>> 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.

Lecture Overview

- Collections
  - Lists
  - Sets
  - Tuples
  - Dictionaries
- File I/O

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 bit…

0	1	1	0	0	0	1	1	0	1	1	0	0	0	0	1	0	1	1	1	0	1	0	0

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

0	1	1	0	0	0	1	1	0	1	1	0	0	0	0	1	0	1	1	1	0	1	0	0

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

0	1	1	0	0	0	1	1	0	1	1	0	0	0	0	1	0	1	1	1	0	1	0	0

C **a** **t**

Files and Filenames

- A file object represents data on your disk drive
  - Can read from it and write to it
- A filename (usually a string) states where to find the data on your disk drive
  - Can be used to find/create a file

- Each operating system comes with its own file system for creating and accessing files:
  - Linux/Mac: `"/home/rea/bbm101/lectures/file_io.pptx"`
  - Windows: `"C:\Users\rea\MyDocuments\cute_dog.jpg"`

Two Types of Filenames

- An Absolute filename gives a specific location on disk: `"/home/rea/bbm101/14wi/lectures/file_io.pptx"` or `"C:\Users\rea\MyDocuments\homework3\images\Husky.png"`
  - Starts with "/" (Unix) or "C:\" (Windows)
  - Warning: code will fail to find the file if you move/rename files or run your program on a different computer
- A Relative filename gives a location relative to the *current working directory*: `"lectures/file_io.pptx"` or `"images\Husky.png"`
  - Warning: code will fail to find the file unless you run your program from a directory that contains the given contents
- *A relative filename is usually a better choice*

Examples

Linux/Mac: These *could* all refer to the same file:

```
"/home/rea/class/140/homework3/images/Husky.png"
"homework3/images/Husky.png"
"images/Husky.png"
"Husky.png"
```

Windows: These *could* all refer to the same file:

```
"C:\Users\rea\My Documents\class\140\homework3\images\Husky.png"
"homework3\images\Husky.png"
"images\Husky.png"
"Husky.png"
```

Depending on what your current working directory is
$ pwd -> print working directory

Locating files: the `os` module

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/Users/r2d2/'

>>> os.listdir()
['death_star_plans', 'princess_leia']

>>> os.listdir('princess_leia')
['Obi-Wan.txt', 'Anakin.txt']

>>> os.chdir('princess_leia')
>>> cwd
'/Users/r2d2/princess_leia'
```

Locating files: the `os` module

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/Users/r2d2/'

>>> os.listdir()
['death_star_plans', 'princess_leia']

>>> os.listdir('princess_leia')
['Obi-Wan.txt', 'Anakin.txt']

>>> os.chdir('princess_leia')
>>> cwd
'/Users/r2d2/princess_leia'
```

`os` module lets us interact with the operating system.
https://docs.python.org/3.6/library/os.html

Locating files: the os module

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/Users/r2d2/'

>>> os.listdir()
['death_star_plans', 'princess_leia']

>>> os.listdir('princess_leia')
['Obi-Wan.txt', 'Anakin.txt']

>>> os.chdir('princess_leia')
>>> cwd
'/Users/r2d2/princess_leia'
```

`os` module lets us interact with the operating system.
https://docs.python.org/3.6/library/os.html

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

Locating files: the os module

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/Users/r2d2/'

>>> os.listdir()
['death_star_plans', 'princess_leia']

>>> os.listdir('princess_leia')
['Obi-Wan.txt', 'Anakin.txt']

>>> os.chdir('princess_leia')
>>> cwd
'/Users/r2d2/princess_leia'
```

`os` module lets us interact with the operating system. https://docs.python.org/3.6/library/os.html

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

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

Locating files: the os module

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/Users/r2d2/'
```

os module lets us interact with the operating system.
https://docs.python.org/3.6/library/os.html

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

```
>>> os.listdir()
['death_star_plans', 'princess_leia']

>>> os.listdir('princess_leia')
['Obi-Wan.txt', 'Anakin.txt']
```

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

```
>>> os.chdir('princess_leia')
>>> cwd
'/Users/r2d2/princess_leia'
```

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

Locating files: the `os` module

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/Users/r2d2/'

>>> os.listdir()
['death_star_plans', 'princess_leia']

>>> os.listdir('princess_leia')
['c3po', 'Obi-Wan.txt', 'Anakin.txt']

>>> os.path.abspath('princess_leia/Obi-Wan.txt')
'/Users/r2d2/princess_leia/Obi-Wan.txt'
```

Locating files: the os module

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/Users/r2d2/'

>>> os.listdir()
['death_star_plans', 'princess_leia']

>>> os.listdir('princess_leia')
['c3po', 'Obi-Wan.txt', 'Anakin.txt']

>>> os.path.abspath('princess_leia/Obi-Wan.txt')
'/Users/r2d2/princess_leia/Obi-Wan.txt'
```

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

Locating files: the os module

```
>>> import os
>>> os.chdir('/Users/r2d2')
>>> os.listdir('princess_leia')
['c3po', 'Obi-Wan.txt', 'Anakin.txt']

>>> os.path.exists('princess_leia/Anakin.txt')
True

>>> os.path.exists('princess_leia/JarJarBinks.txt')
False

>>> os.path.isdir('princess_leia/c3po')
True

>>> os.path.isdir('princess_leia/Obi-Wan.txt’)
False
```

Locating files: the os module

```
>>> import os
>>> os.chdir('/Users/r2d2')
>>> os.listdir('princess_leia')
['c3po', 'Obi-Wan.txt', 'Anakin.txt']

>>> os.path.exists('princess_leia/Anakin.txt')
True

>>> os.path.exists('princess_leia/JarJarBinks.txt')
False

>>> os.path.isdir('princess_leia/c3po')
True

>>> os.path.isdir('princess_leia/Obi-Wan.txt’)
False
```

Check whether or not a file/directory exists.

Locating files: the os module

```
>>> import os
>>> os.chdir('/Users/r2d2')
>>> os.listdir('princess_leia')
['c3po', 'Obi–Wan.txt', 'Anakin.txt']

>>> os.path.exists('princess_leia/Anakin.txt')
True
```

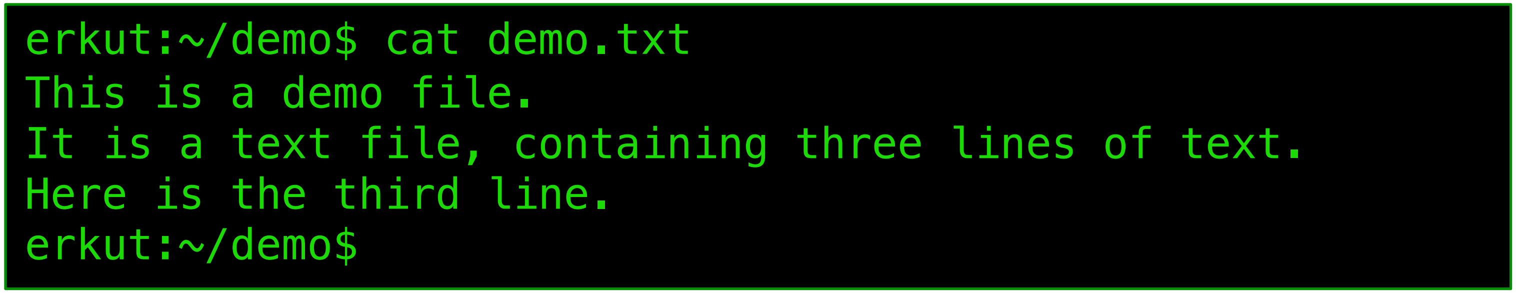
Check whether or not a file/directory exists.

```
>>> os.path.exists('princess_leia/JarJarBinks.txt')
False

>>> os.path.isdir('princess_leia/c3po')
True
```

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

```
>>> os.path.isdir('princess_leia/Obi–Wan.txt’)
False
```

Reading files

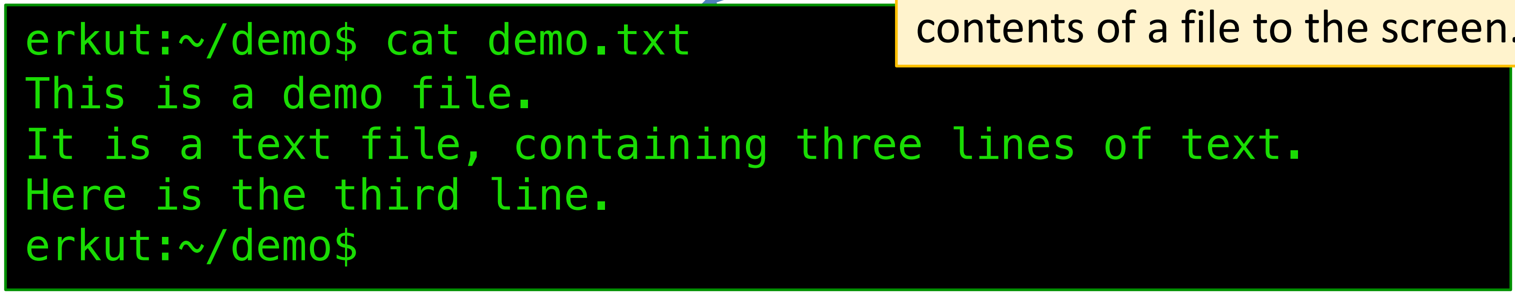


```
erkut:~/demo$ cat demo.txt
This is a demo file.
It is a text file, containing three lines of text.
Here is the third line.
erkut:~/demo$
```

```
>>> f = open('demo.txt')
>>> type(f)
<type 'file'>

>>> f.readline()
'This is a demo file.\n'
```

Reading files

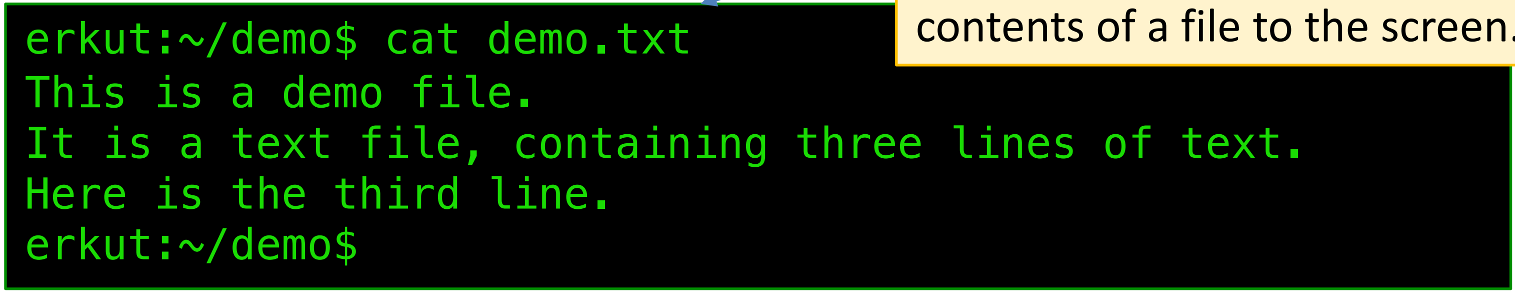
This is the command line. We'll see lots more about this later, but for now, it suffices to know that the command cat prints the contents of a file to the screen.



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

```
>>> f = open('demo.txt')
>>> type(f)
<type 'file'>

>>> f.readline()
'This is a demo file.\n'
```

Reading files



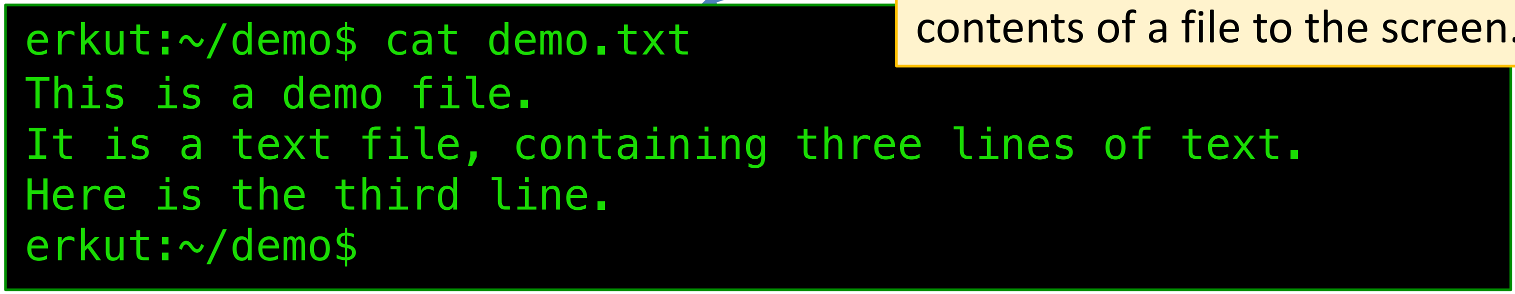
```
erkut:~/demo$ cat demo.txt
This is a demo file.
It is a text file, containing three lines of text.
Here is the third line.
erkut:~/demo$
```

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.

```
>>> f = open('demo.txt')
>>> type(f)
<type 'file'>

>>> f.readline()
'This is a demo file.\n'
```

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

Reading files



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

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.

```
>>> f = open('demo.txt')
>>> type(f)
<type 'file'>
```

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

```
>>> f.readline()
'This is a demo file.\n'
```

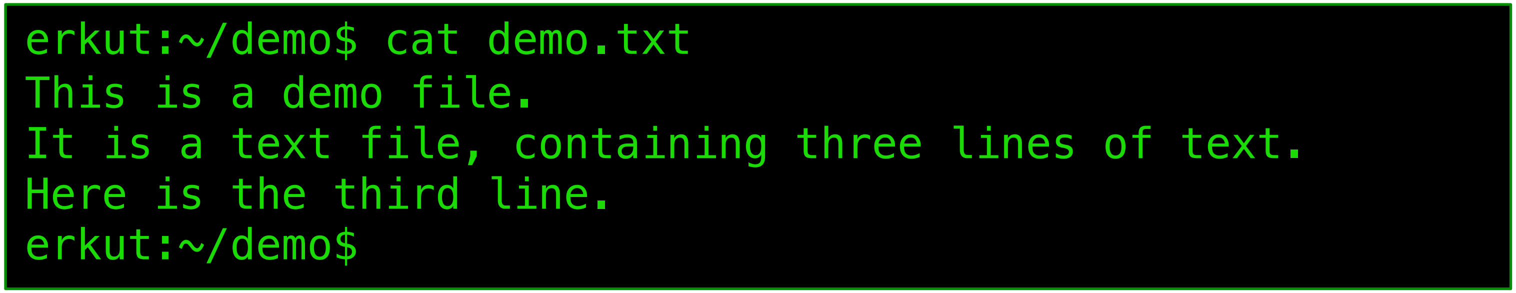
Provides a method for reading a single line from the file. The string `'\n'` is a **special character** that represents a new line. More on this soon.

Reading files

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

```
>>> f = open('demo.txt')
>>> f.readline()
'This is a demo file.\n'

>>> f.readline()
'It is a text file, containing three lines of text.\n'

>>> f.readline()
'Here is the third line.\n'

>>> f.readline()
```

Reading files



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

```
>>> f = open('demo.txt')
>>> f.readline()
'This is a demo file.\n'

>>> f.readline()
'It is a text file, containing three lines of text.\n'

>>> f.readline()
'Here is the third line.\n'

>>> f.readline()
```

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

Reading files

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

```
>>> f = open('demo.txt')
>>> f.readline()
'This is a demo file.\n'

>>> f.readline()
'It is a text file, containing three lines of text.\n'

>>> f.readline()
'Here is the third line.\n'

>>> 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

```
>>> f = open('demo.txt')
>>> for line in f:
...     for wd in line.split():
...         print(wd.strip('.,'))
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

Reading files

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

```
>>> f = open('demo.txt')
>>> for line in f:
...     for wd in line.split():
...         print(wd.strip('.,'))
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

Reading files

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).

```
>>> f = open('demo.txt')
>>> for line in f:
...     for wd in line.split():
...         print(wd.strip('.,'))
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

Reading files

```
>>> f = open('demo.txt')
>>> for line in f:
...     for wd in line.split():
...         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.

Reading files

```
>>> f = open('demo.txt')
>>> for line in f:
...     for wd in line.split():
...         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

```
>>> with open('demo.txt’) as f:
...     for line in f:
...         for wd in line.split():
...             print(wd.strip('.,'))
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

Reading files

```
>>> with open('demo.txt’) as f:
...     for line in f:
...         for wd in line.split():
...             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. It suffices to know that this is equivalent to what we did on the previous slide.

Reading files

```
>>> with open('demo.txt’) as f:
...     for line in f:
...         for wd in line.split():
...             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. 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.

Reading a File Example

```python
# Count the number of words in a text file
in_file = "thesis.txt"
myfile = open(in_file)
num_words = 0
for line_of_text in myfile:
    word_list = line_of_text.split()
    num_words += len(word_list)
myfile.close()

print("Total words in file: ", num_words)
```

Reading a File Multiple Times

You can iterate over a **list** as many times as you like:

```
mylist = [ 3, 1, 4, 1, 5, 9 ]
for elt in mylist:
  … process elt
for elt in mylist:
  … process elt
```

Iterating over a **file** uses it up:

```
myfile = open("datafile.dat")
for line_of_text in myfile:
    … process line_of_text
for line_of_text in myfile:
    … process line_of_text
```

This loop body will never be executed!

How to read a file multiple times?

Solution 1: Read into a list, then iterate over it

```
myfile = open("datafile.dat")
mylines = []
for line_of_text in myfile:
  mylines.append(line_of_text)
… use mylines
```

Solution 2: Re-create the file object (slower, but a better choice if the file does not fit in memory)

```
myfile = open("datafile.dat")
for line_of_text in myfile:
   … process line_of_text
myfile = open("datafile.dat")
for line_of_text in myfile:
   … process line_of_text
```

Writing files

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

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for reading

>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
```

Writing files

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

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

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for reading

>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
```

Writing files

```
>>> f = open('animals.txt', 'w')
```

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

```
>>> f.read()
```

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

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for reading

>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
```

Writing files

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

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

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

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for reading
```

```
>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
```

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.

Writing files

```
>>> f = open('animals.txt', 'w')
>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
>>> f.close()

>>> f = open('animals.txt', 'r')
>>> for line in f:
...     print(line, end=’’)
cat
dog
bird
goat
```

Writing files

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

```
>>> f = open('animals.txt', 'w')
>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
>>> f.close()

>>> f = open('animals.txt', 'r')
>>> for line in f:
...     print(line, end=’’)
cat
dog
bird
goat
```

Writing files

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

```
>>> f = open('animals.txt', 'w')
>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
>>> f.close()
```

Each write appends to the end of the file.

```
>>> f = open('animals.txt', 'r')
>>> for line in f:
...     print(line, end=’’)
cat
dog
bird
goat
```

Writing files

```
>>> f = open('animals.txt', 'w')
>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
>>> 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.

```
>>> f = open('animals.txt', 'r')
>>> for line in f:
...     print(line, end=’’)
cat
dog
bird
goat
```

Writing files

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

```
>>> f = open('animals.txt', 'w')
>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
>>> f.close()
```

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.

```
>>> f = open('animals.txt', 'r')
>>> for line in f:
...     print(line, end=’’)
cat
dog
bird
goat
```

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

Writing files

```
>>> f = open('animals.txt', 'w')
>>> f.write('cat\n')
>>> f.write('dog\n')
>>> f.write('bird\n')
>>> f.write('goat\n')
>>> 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.

```
>>> f = open('animals.txt', 'r')
>>> for line in f:
...     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.

More Examples - 1

```
nameHandle = open('characters.txt', 'w')
for i in range(2):
    name = input('Enter name: ')
    nameHandle.write(name + '\n')
nameHandle.close()
```

```
nameHandle = open('characters.txt', 'r')
for line in nameHandle:
    print(line)
nameHandle.close()
```

- If we had typed in the names Rick and Morty, this will print

```
Rick

Morty
```

- The extra line between `Rick` and `Morty` is there because print starts a new line each time it encounters the `'\n'` at the end of each line in the file.

More Examples - 2

```
nameHandle = open('characters.txt', 'w')
nameHandle.write('Jerry\n')
nameHandle.write('Beth\n')
nameHandle.close()

nameHandle = open('characters.txt', 'r')
for line in nameHandle:
   print line[:-1]
nameHandle.close()
```

- It will print

  ```
  Jerry
  Beth
  ```

- Notice that
  - we have overwritten the previous contents of the file.
  - `print line[:-1]` avoids extra newline in the output

More Examples - 3

```
nameHandle = open('characters.txt', 'a')
nameHandle.write('Rick\n')
nameHandle.write('Morty\n')
nameHandle.close()
```

```
nameHandle = open(characters.txt', 'r')
for line in nameHandle:
   print line[:–1]
nameHandle.close()
```

- It will print

```
Jerry
Beth
Rick
Morty
```

- Notice that we can open the file for appending (instead of writing) by using the argument `'a'`.

Common functions for accessing files

- `open(fn, 'w')` `fn` is a string representing a file name. Creates a file for writing and returns a file handle.
- `open(fn, 'r')` `fn` is a string representing a file name. Opens an existing file for reading and returns a file handle.
- `open(fn, 'a')` `fn` is a string representing a file name. Opens an existing file for appending and returns a file handle.
- `fn.close()` closes the file associated with the file handle `fn`.

Common functions for accessing files

- `fn.read()` returns a string containing the contents of the file associated with the file handle `fn`.
- `fn.readline()` returns the next line in the file associated with the file handle `fn`.
- `fn.readlines()` returns a list each element of which is one line of the file associated with the file handle `fn`.
- `fn.write(s)` write the string s to the end of the file associated with the file handle `fn`.
- `fn.writelines(S)` S is a sequence of strings. Writes each element of S to the file associated with the file handle `fn`.

Formatting Strings

```
>>> x = 23
>>> print(‘x = %d' % x)
x = 23

>>> animal = 'unicorn'
>>> print('My pet %s' % animal)
My pet unicorn

>>> x=2.718; y=1.618
>>> print('%f divided by %f is %f' % (x,y,x/y))
2.718000 divided by 1.618000 is 1.679852

>>> print('%.3f divided by %.3f is %.8f' % (x,y,x/y))
2.718 divided by 1.618 is 1.67985167
```

Formatting Strings

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

```
>>> x = 23
>>> print(‘x = %d' % x)
x = 23

>>> animal = 'unicorn'
>>> print('My pet %s' % animal)
My pet unicorn

>>> x=2.718; y=1.618
>>> print('%f divided by %f is %f' % (x,y,x/y))
2.718000 divided by 1.618000 is 1.679852

>>> print('%.3f divided by %.3f is %.8f' % (x,y,x/y))
2.718 divided by 1.618 is 1.67985167
```

Formatting Strings

```
>>> x = 23
>>> print(‘x = %d' % x)
x = 23

>>> animal = 'unicorn'
>>> print('My pet %s' % animal)
My pet unicorn

>>> x=2.718; y=1.618
>>> print('%f divided by %f is %f' % (x,y,x/y))
2.718000 divided by 1.618000 is 1.679852

>>> 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

Formatting Strings

```
>>> x = 23
>>> print(‘x = %d' % x)
x = 23

>>> animal = 'unicorn'
>>> print('My pet %s' % animal)
My pet unicorn

>>> x=2.718; y=1.618
>>> print('%f divided by %f is %f' %
2.718000 divided by 1.618000 is 1.67

>>> 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.

Formatting Strings

```
>>> x = 23
>>> print(‘x = %d' % x)
x = 23
```

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

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

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

```
>>> x=2.718; y=1.618
>>> print('%f divided by %f is %f' %
2.718000 divided by 1.618000 is 1.67
```

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

```
>>> print('%.3f divided by %.3f is %.8f' % (x,y,x/y))
2.718 divided by 1.618 is 1.67985167
```

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

Formatting Strings

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

```
>>> x=2.718; y=1.618
>>> print('%f divided by %f is %f' % (x,y,x/y,1.0))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:not all arguments converted during string formatting
```

```
>>> x=2.718; y=1.618
>>> print('%f divided by %f is %f' % (x,y))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format string
```

Writing modules

- Python provides modules (e.g., math, os, time)
- But we can also write our own, and import from them with same syntax

```
>>> import prime
>>> prime.is_prime(2)
True

>>> prime.is_prime(3)
True

>>> prime.is_prime(1)
False

>>> prime.is_prime(33)
False
```

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

```
>>> from prime import *
>>> is_prime(7)
True

>>> is_square(7)
False

>>> is_prime(373)
True
```

```
import math                   prime.py

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

def is_square(n):
       r = int(math.sqrt(n))
       return(r*r==n or (r+1)*(r+1)==n)
```

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`

```
>>> from prime import *
>>> is_prime(7)
True

>>> is_square(7)
False

>>> is_prime(373)
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

def is_square(n):
        r = int(math.sqrt(n))
        return(r*r==n or (r+1)*(r+1)==n)
```

Writing modules

```
>>> from prime import *
>>> is_prime(7)
True

>>> is_square(7)
False

>>> is_prime(373)
True
```

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

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

def is_square(n):
        r = int(math.sqrt(n))
        return(r*r==n or (r+1)*(r+1)==n)
```

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