Wikimedia Commons

BBM 101

Introduction to Programming I

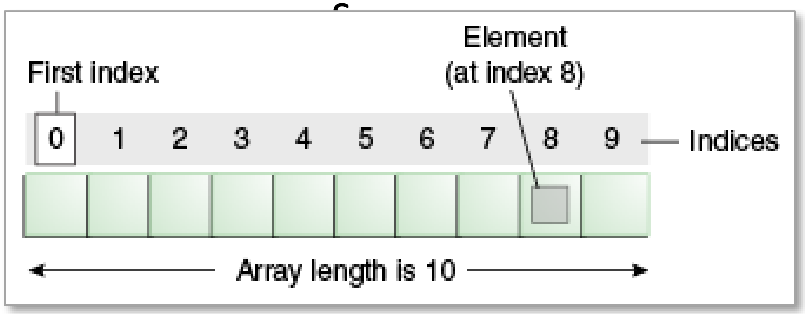
Lecture #07 – Tuples, Sets, Dictionaries

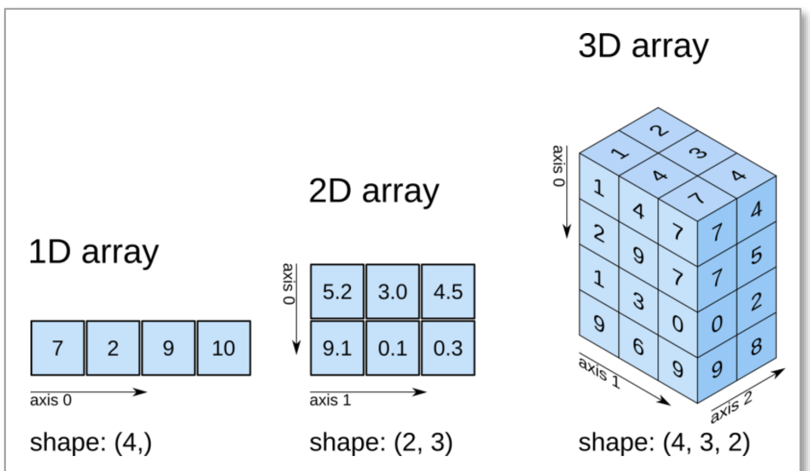
HACETTEPE UNIVERSITY

Tunca Doğan, Fuat Akal & Aydın Kaya // Fall 2023

Last time... **Arrays, Lists**

Arrays





Lists

```
>>> list1 = [1, 2, 3]
>>> list1.append(4)
>>> list1.insert(2, 5)
>>> list2 = [10, 20]
>>> list1.extend(list2)

>>> list1.append(list2)
```

Lecture Overview

- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries

Disclaimer: Much of the material and slides for this lecture were borrowed from
—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class
—Keith Levin's University of Michigan STATS 507 class

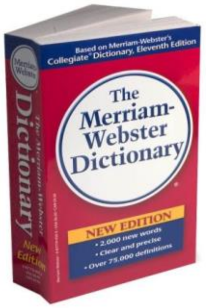
Recall: Data Structures

- 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



Lecture Overview

- Arrays
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries

Disclaimer: Much of the material and slides for this lecture were borrowed from
—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class

Tuples

- Like strings, **tuples** are ordered sequences of elements.
- The individual elements can be of any type, and need not be of the same type as each other.
- Literals of type tuple are written by enclosing a comma-separated list of elements within parentheses.
- Tuples differ from lists in one hugely important way:
  - Lists are mutable. In contrast, tuples are immutable.

- ```
  t1 = ()
  t2 = (1, 'two', 3)
  print(t1)
  print(t2)

  >> ()
  >> (1, 'two', 3)
  ```
  ```

Tuples are Sequences

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

>> t[1:4]
('b', 'c', 'd')

>> t[-1]
'e'

>> len(t)
5
```

As sequences, tuples support indexing, slices, etc.

And of course, sequences have a length.

Reminder: sequences support all the operations listed here: https://docs.python.org/3.3/library/stdtypes.html#typesseq

Tuples are Sequences

- Like strings, tuples can be concatenated, indexed, and sliced.

- ```
  t1 = (1, 'two', 3)
  t2 = (t1, 3.25)
  print(t2)
  print((t1 + t2))
  print((t1 + t2)[3])
  print((t1 + t2)[2:5])
  ```

  ```
  >> ((1, 'two', 3), 3.25)
  >> (1, 'two', 3, (1, 'two', 3), 3.25)
  >> (1, 'two', 3)
  >> (3, (1, 'two', 3), 3.25)
  ```
  ```

Tuples are Sequences

- Even if we can’t modify the elements of a tuple, we can make a variable reference a new tuple holding different information.

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")

print(julia[2])
print(julia[2:6])
print(len(julia))
```

```
1967
(1967, 'Duplicity', 2009, 'Actress')
7
```

```
for field in julia:
    print(field)

julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]
print(julia)
```

```
('Julia', 'Roberts', 1967, 'Eat Pray Love', 2010, 'Actress',
'Atlanta, Georgia')
```

THE PYTHON TUTOR

Tuples and Mutability

- Unlike lists, tuples are immutable.
- As with strings, if we try to use item assignment to modify one of the elements of the tuple, we get an error.

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")

julia[0] = 'X'        # not allowed
TypeError: 'tuple' object does not support item assignment
```

Tuple Comparison

Tuples support comparison, which works analogously to string ordering.

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

>>(2,2,20) <= (2,2,2)
False
```

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

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

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

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

>> (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.

Tuple Assignment

- Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

```
>> julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")

>> (name, surname, birth_year, movie, movie_year, profession, birth_place) = julia
```

- Naturally, the number of variables on the left and the number of values on the right have to be the same.

```
>> (a, b, c, d) = (1, 2, 3)
ValueError: need more than 3 values to unpack
```

Tuple Assignment

```
>> a = 10
>> b = 5
>> 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)
5 10
```

Common pattern: swap the values of two variables.

```
>> a = 10
>> b = 5
>> (a, b) = (b, a)
>> 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.

Tuple Assignment

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

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

```
>> (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.

```
>> (x, y, z) = ('a', 'b')
ValueError: not enough values to unpack (expected 3, got 2)
```

Tuple Assignment

```
>> email = 'bbm101@cs.hacettepe.edu.tr'
>> email.split('@')
['bbm101', 'cs.hacettepe.edu.tr']
```

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

```
>> (user, domain) = email.split('@')
>> user
'bbm101'
>> domain
'cs.hacettepe.edu.tr'
```

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, ['bbm101','cs.hacettepe.edu,tr'] .

```
>> (x, y, z) = ('cat')
>> 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

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

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

```
>> t = divmod(13, 4)
>> t
(3, 1)

>> (quotient, remainder) = divmod(13, 4)
>> quotient
3
>> 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

Tuples as Return Values

- A for statement can be used to iterate over the elements of a tuple.

- ```
  def findDivisors (n1, n2):
          """Assumes n1 and n2 are positive ints
              Returns a tuple containing all common divisors
              of n1 & n2"""
          divisors = () #the empty tuple
          for i in range(1, min (n1, n2) + 1):
              if n1%i == 0 and n2%i == 0:
                  divisors = divisors + (i,)
          return divisors

  divisors = findDivisors(20, 100)
  print(divisors)
  total = 0
  for d in divisors:
          total += d
  print(total)
  ```

To create a tuple with a single element, you have to include the final comma

```
>> (1, 2, 4, 5, 10, 20)
>> 42
```
```

Variable-length Arguments

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

>> my_min(4,5,6,10)
4

>> def print_all( *args):
...    print(args)
>> print_all('cat', 'dog', 'bird')
('cat', 'dog', 'bird')

>> 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.

Gather and Scatter

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

```
>> t = (13, 4)
>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

divmod takes two arguments, so this is an error.

```
>> 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.

```
>> *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

```
>> t1 = ['apple', 'orange', 'banana', 'kiwi']
>> t2 = [1, 2, 3, 4]
>> zip(t1, t2)
<zip at 0x10c95d5c8>

>> for tup in zip(t1,t2):
...    print(tup)
('apple', 1)
('orange', 2)
('banana', 3)
('kiwi', 4)
```

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

Iterators are, in essence, objects that support for-loops. All sequences are iterators. Iterators support, crucially, a method __next__(), which returns the "next element".
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.

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

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

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

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

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

```
>> def count_matches(s, t):
...    cnt = 0
...    for (a,b) in zip(s,t):
...        if a==b:
...            cnt += 1
...    return(cnt)

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

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

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

Related function: enumerate()

```
>> for t in enumerate('goat'):
...    print(t)
(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.
https://docs.python.org/3/library/functions.html#enumerate

```
>> s = 'goat'
>> for i in range(len(s)):
...    print((i,s[i]))
(0, 'g')
(1, 'o')
(2, 'a')
(3, 't')
```

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

Lecture Overview

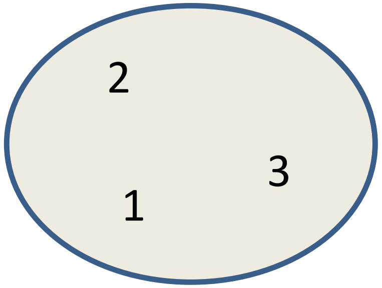
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries

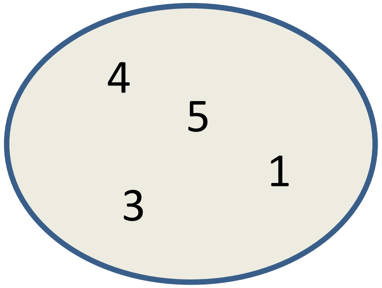
Sets

- 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 ∈ Python: in 4 in prime ⇒ False
- union ∪ Python: | odd | prime ⇒ { 1, 2, 3, 5 }
- intersection ∩ Python: & odd & prime ⇒ { 3, 5 }
- difference \ or - Python: - odd – prime ⇒ { 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 <u>not</u> 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

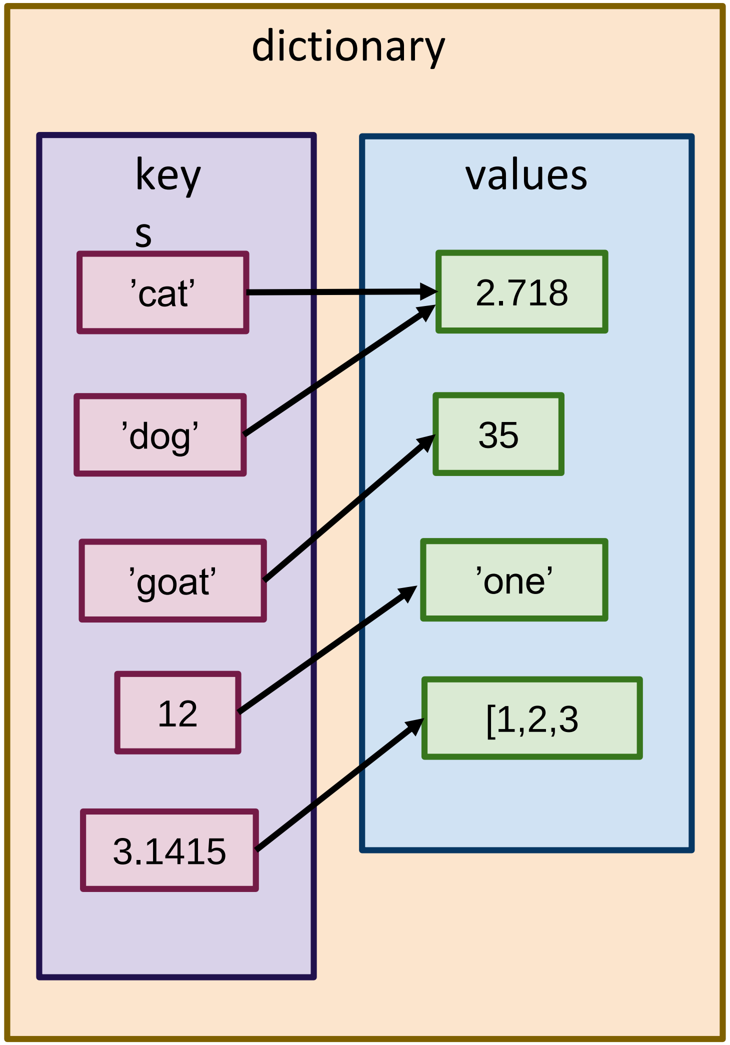
```
list1 in myset                       ⇒ ???
list3 in myset                       ⇒ ???
```

Lecture Overview

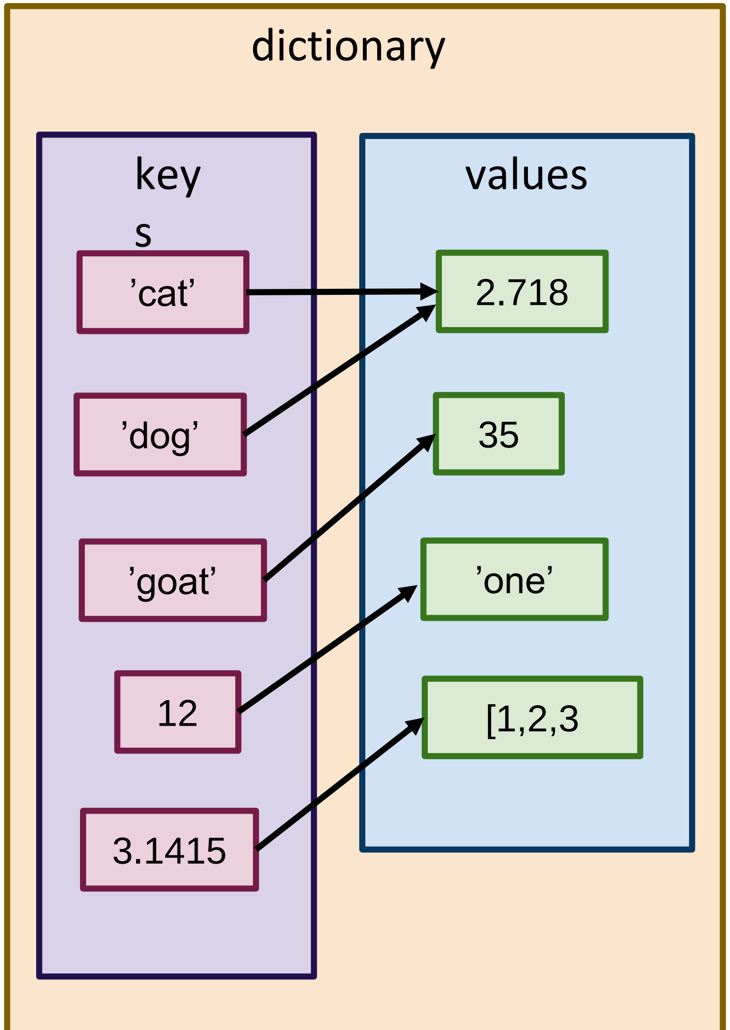
- Collections
  - Lists
  - Tuples
  - Sets
  - Dictionaries

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



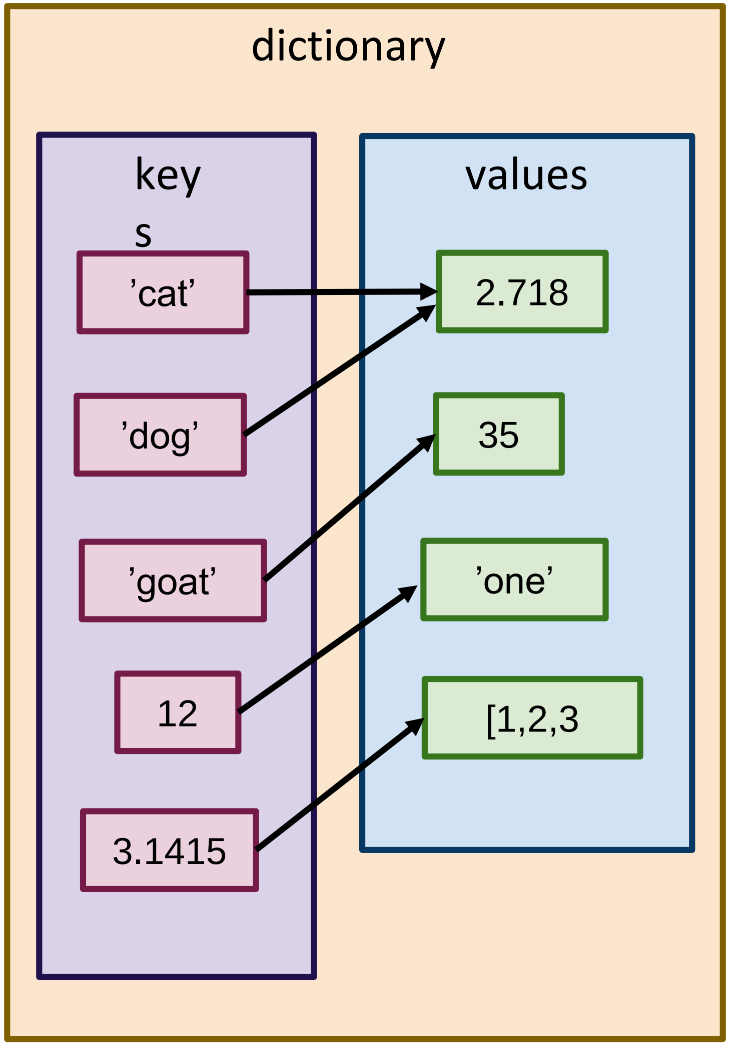
Dictionaries



- 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

Example:
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'}

>>> len(huceng2name)
4

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

Length of a dictionary is just the number of items.

Empty dictionary has length 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
0.00022196769714355 47
```

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

>>> start_time = time.time()
>>> 8675309 in dict_of_numbers
>>> time.time() – start_time
0.000219676971435547
```

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.

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
0.00022196769714355469
```

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
0.00021967697143554 7
```

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

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

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

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.