DICTIONARY

Dictionary vs. multi-way `if` statement

Uses of a dictionary:

- container with custom indexes
- alternative to the multi-way `if` statement

```
def complete(abbreviation):
   ‘’’returns day of the week
corresponding to abbreviation‘’’
         days = {
                 'Mo': 'Monday',
                 'Tu':'Tuesday',
                 'We': 'Wednesday',
                 'Th': 'Thursday',
                 'Fr': 'Friday',
                 'Sa': 'Saturday',
                 'Su':'Sunday‘
                 }

    return days[abbreviation]
```

```
def complete(abbreviation):
    ‘’’returns day of the week
corresponding to abbreviation‘’’

    if abbreviation == 'Mo':
        return 'Monday'
    elif abbreviation == 'Tu':
        return 'Tuesday'
    elif
        ......
    else: # abbreviation must be Su
        return 'Sunday'
```

Exercise

- A company sells five types of items, labeled a through e. One item of type a costs $2, one item of type b costs $3, items of type c and d cost $4 and each item of type e costs $5.
- Write a function called invoice that takes as input the type of item purchased and the number of items of that type purchased and returns the total cost.
- Use a dictionary in your function

Count Words

- Need to count how many words with the same length were in a given text
- Simple to generate the list of length of words
- Now how do I keep track of length that are in the list multiple times?

Dictionary as a container of counters

Uses of a dictionary:

- container with custom indexes
- alternative to the multi-way `if` statement
- container of counters

Problem: computing the number of occurrences of items in a list

```
>>> grades = [95, 96, 100, 85, 95, 90, 95, 100, 100]
>>> frequency(grades)
{96: 1, 90: 1, 100: 3, 85: 1, 95: 3}
>>>
```

Solution: Iterate through the list and, for each grade, increment the counter corresponding to the grade.

Problems:

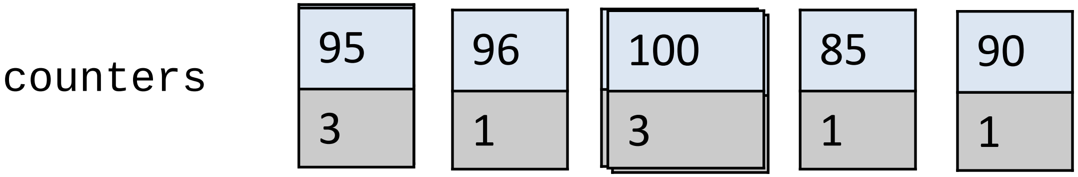
- impossible to create counters before seeing what’s in the list
- how to store grade counters so a counter is accessible using the corresponding grade

Solution: a dictionary mapping a grade (the key) to its counter (the value)

Dictionary as a container of counters

Problem: computing the number of occurrences of items in a list

```
>>> grades = [95, 96, 100, 85, 95, 90, 95, 100, 100]
                   ^    ^       ^     ^     ^ ^   ^ ^    ^
```



```
def frequency(itemList):
    'returns frequency of items in itemList'

    counters = {}
    for item in itemList:
        if item in counters: # increment item counter
            counters[item] += 1
        else: # create item counter
            counters[item] = 1
    return counters
```

Exercise

Implement function wordcount() that takes as input a text—as a string— and prints the frequency of each word in the text; assume there is no punctuation in the text.

```
>>> text = 'all animals are equal but some animals are more equal than
other'
>>> wordCount(text)
all      appears 1 time.
animals  appears 2 times.
some     appears 1 time.
equal    appears 2 times.
but      appears 1 time.
other    appears 1 time.
are      appears 2 times.
than     appears 1 time.
more     appears 1 time.
>>>
```

```
print('Sorted by key')
    for (key, value) in sorted(counters.items()):
        print(key, counters[key])
```

```
>>> help (sorted)
Help on built-in function sorted in module builtins:

sorted(iterable, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customise the sort order, and the
    reverse flag can be set to request the result in descending order.
```

```
def wordCount(text):
    'prints frequency of each word in text'

    wordList = text.split()  # split text into list of words

    counters ={}             # dictionary of counters
    for word in wordList:
        if word in counters: # counter for word exists
            counters[word] += 1
        else:                # counter for word doesn't exist
            counters[word] = 1

    for word in counters:    # print word counts
        if counters[word] == 1:
            print('{:8} appears {} time.'.format(word, counters[word]))
        else:
            print('{:8} appears {} times.'.format(word, counters[word]))
```

To remove punctuation:
from string import punctuation #import only the punctuation object in string

transTable =str.maketrans(punctuation, ' '*len(punctuation))
s = s.translate(transTable)

What is s??

TUPLE

tuples

- Syntax:
  - (item, item, item,…)

- Ordered, IMMUTABLE, indexed container
- As a list a tuple can contain heterogenous objects/data types
- In a sense it is the “default” container that Python uses if you do not specify one

Tuples in our midst

```
>>> myList = eval(input(‘Please enter three integers:’))
>>>1,2,3
```

- In this circumstance Python will capture 1,2,3 as a tuple (1,2,3)
- Lists of items entered without [ ] (list) or ( ) (tuple) are by default treated as tuples!

What am i?

List

```
>>> whatAmI=[1,2,3]
>>> type(whatAmI)
<class 'list'>
```

tuple

```
>>> whatAmI=(1,2,3)
>>> type(whatAmI)
<class 'tuple'>
>>> whatAmI=1,2,3
>>> type(whatAMI)
<class 'tuple'>
```

Built-in class `tuple`

The class `tuple` is very similar to class list … except that it is immutable

```
>>> lst = ['one', 'two', 3]
>>> lst[2]
3
>>> lst[2] = 'three'
>>> lst
['one', 'two', 'three']
>>> tpl = ('one', 'two', 3)
>>> tpl
('one', 'two', 3)
>>> tpl[2]
3
>>> tpl[2] = 'three'
Traceback (most recent call last):
  File "<pyshell#131>", line 1, in <module>
    tpl[2] = 'three'
TypeError: 'tuple' object does not support item assignment
>>>
```

Exercise

- Build a tuple containing the numbers 1,2,3 and the strings ‘a’, ‘b’, ‘c’
- Compute the length of the tuple
- Pass the whole tuple as an argument to the print function
- Print all elements of the tuple, one on each line, with no commas.
- Turn this tuple into a list without retyping the elements and without using a for loop

Why bother?

- Why do we need tuples?
- Sometimes we want immutable lists
- For example if we want to create a dictionary with more than one element as a key…
- Example:
  - A phone book that needs to allow me to search by first and/or last name

Example

Implement function `lookup()` that implements a phone book lookup application. Your function takes, as input, a dictionary representing a phone book, mapping tuples (containing the first and last name) to strings (containing phone numbers)

```
>>> phonebook = {
         ('Anna','Karenina'):'(123)456-78-90',
         ('Yu', 'Tsun'):'(901)234-56-78',
         ('Hans', 'Castorp'):'(321)908-76-54'}
>>> lookup(phonebook)
Enter the first name: Anna
Enter the last name: Karenina
(123)456-78-90
Enter the first name:
```

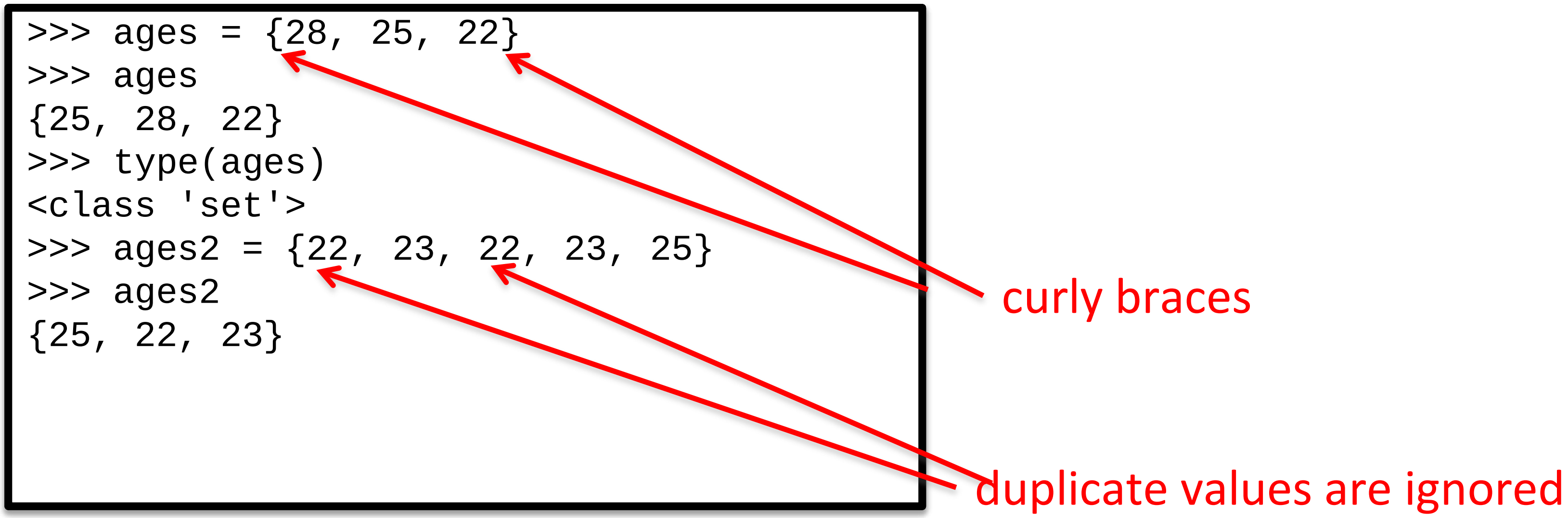
```
def lookup(phonebook):
    '''implements interactive phone book service using the input
       phonebook dictionary'''
    while True:
        first = input('Enter the first name: ')
        last = input('Enter the last name: ')

        person = (first, last)# construct the key

        if person in phonebook:# if key is in dictionary
            print(phonebook[person] # print value
        else:              # if key not in dictionary
            print('The name you entered is not known.')
```

SET

Built-in class set

The built in class set represents a mathematical set

- an unordered collection of non-identical items
- supports operations such as set membership, set union, set intersection, set difference, etc



```
>>> ages = {28, 25, 22}
>>> ages
{25, 28, 22}
>>> type(ages)
<class 'set'>
>>> ages2 = {22, 23, 22, 23, 25}
>>> ages2
{25, 22, 23}
```

curly braces

duplicate values are ignored

Leveraging sets

- Sets can be useful to eliminate duplicates from a list
- One can translate a list with duplicates into a set
- Then back from a set to a list

```
>>> myList=[1,1,2,3,3,4,5]
>>> noDuplicates = set(myList)
>>> noDuplicates
{1, 2, 3, 4, 5}
>>> myList=list(noDuplicates)
>>> myList
[1, 2, 3, 4, 5]
```

- Or simply

```
myList =list(set(myList))
```

set operators

Operation	Explanation
`s == t`	`True` if sets `s` and `t` contain the same elements, `False` otherwise
`s != t`	`True` if sets `s` and `t` do not contain the same elements, `False` otherwise
`s <= t`	`True` if every element of set `s` is in set `t`, `False` otherwise
`s < t`	`True` if `s` <= `t` and `s` != `t`
`s | t`	Returns the union of sets `s` and `t`
`s & t`	Returns the intersection of sets `s` and `t`
`s - t`	Returns the difference between sets `s` and `t`
`s ^ t`	Returns the symmetric difference of sets `s` and `t`

```
>>> ages
{28, 25, 22}
>>> ages2
{25, 22, 23}
>>> 28 in ages
True
>>> len(ages2)
3
>>> ages == ages2
False
>>> {22, 25} < ages2
True
>>> ages <= ages2
False
>>> ages | ages2
{22, 23, 25, 28}
>>> ages & ages2
{25, 22}
>>> ages - ages2
{28}
>>> ages ^ ages2
{28, 23}
>>>ages[0]
Traceback (most recent call
last):
  File "<pyshell#4>", line 1,
in <module>
    ages[0]
TypeError: 'set' object does
not support indexing
```

set methods

```
>>> ages
{28, 25, 22}
>>> ages2
{25, 22, 23}
>> ages.add(30)
>>> ages
{25, 28, 30, 22}
>>> ages.remove(25)
>>> ages
{28, 30, 22}
>>> ages.clear()
>>> ages
set()
```

Operation	Explanation
`s.add(item)`	add `item` to set `s`
`s.remove(item)`	remove `item` from set `s`
`s.clear()`	removes all elements from `s`

Note that sets are mutable

Empty set

- Suppose we need an empty set

```
>>> mySet = {}
>>> type(mySet)
<class 'dict'>
```

- Python created an empty DICTIONARY…

```
>>> mySet = set()
>>> type(mySet)
<class 'set'>
```

Exercise

- Fill the table below with Y for yes and N for no describing features of each container class

Data type	ORDERED	MUTABLE	ALLOWS DUPLICATES	ALLOWS RETRIEVAL BY INDEX [ ]	ONE CAN ITERATE OVER ITS ELEMENTS
dict					
list					
set					
tuple					

RANDOM MODULE

Randomness

Some apps need numbers generated “at random” (i.e., from some probability distribution):

- scientific computing
- financial simulations
- cryptography
- computer games

Truly random numbers are hard to generate

Most often, a pseudorandom number generator is used

- numbers only appear to be random
- they are really generated using a deterministic process

The Python standard library module `random` provides a pseudo random number generator as well useful sampling functions

Standard Library module `random`

Function `randrange()` returns a “random” integer number from a given range

Example usage: simulate the throws of a die

Function `uniform()` returns a “random” `float` number from a given range

range is from 1 up to (but not including) 7

```
>>> import random
>>> random.randrange(1, 7)
2
>>> random.randrange(1, 7)
1
>>> random.randrange(1, 7)
4
>>> random.randrange(1, 7)
2
>>> random.uniform(0, 1)
0.19831634437485302
>>> random.uniform(0, 1)
0.027077323233875905
>>> random.uniform(0, 1)
0.8208477833085261
>>>
```

Standard Library module random

Defined in module random are functions `shuffle()`, `choice()`, `sample()`, …

```
>>> names = ['Ann', 'Bob', 'Cal', 'Dee', 'Eve', 'Flo', 'Hal', 'Ike']
>>> import random
>>> random.shuffle(names)
>>> names
['Hal', 'Dee', 'Bob', 'Ike', 'Cal', 'Eve', 'Flo', 'Ann']
>>> random.choice(names)
'Bob'
>>> random.choice(names)
'Ann'
>>> random.choice(names)
'Cal'
>>> random.choice(names)
'Cal'
>>> random.sample(names, 3)
['Ike', 'Hal', 'Bob']
>>> random.sample(names, 3)
['Flo', 'Bob', 'Ike']
>>> random.sample(names, 3)
['Ike', 'Ann', 'Hal']
>>>
```

Random methods

- randrange(start, stop, step=1)
  - Choose a random item from range(start, stop[, step])
- uniform(a, b)
  - Get a random number in the range [a, b)
- sample(population, k)
  - Chooses k unique random elements from a sequence or set.
- choice(seq)
  - Choose a random element from a non-empty sequence.
- shuffle(x) method of Random instance
  - Shuffle list x in place.

ENCODING

Character encodings

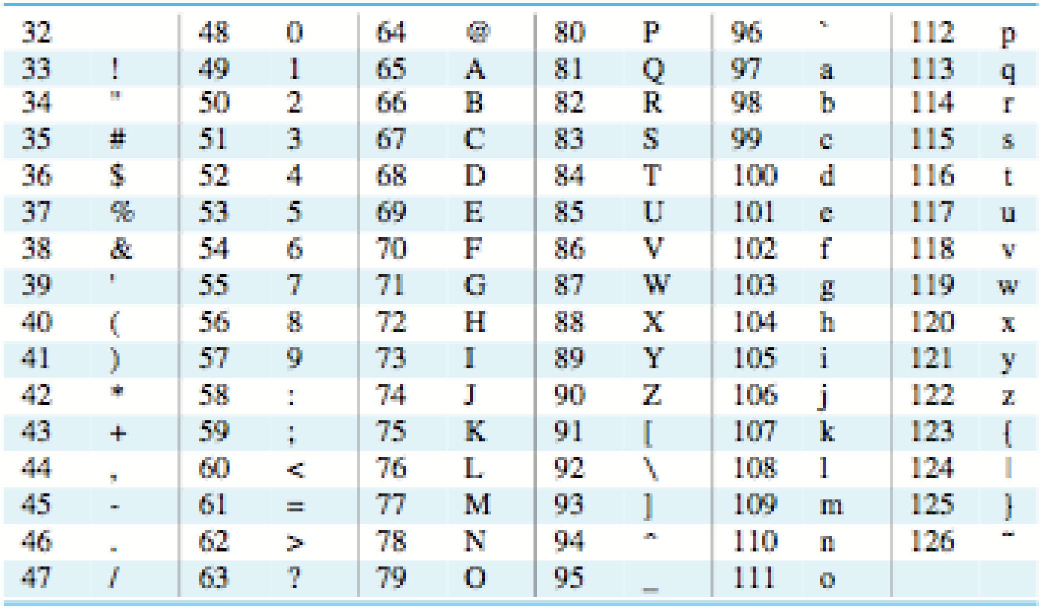
A string (`str`) object contains an ordered sequence of characters which can be any of the following:

- lowercase and uppercase letters in the English alphabet: `a b c … z` and `A B C … Z`
- decimal digits: `0 1 2 3 4 5 6 7 8 9`
- punctuation: `, . : ; ' " ! ?` etc.
- Mathematical operators and common symbols: `= < > + - / * $ # % @ &` etc.
- More later

Each character is mapped to a specific bit encoding, and this encoding maps back to the character.

For many years, the standard encoding for characters in the English language was the American Standard Code for Information Interchange (ASCII)

ASCII



32		48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		

The code for a is 97, which is 01100001 in binary or 0x61 in hexadecimal notation

For many years, the standard encoding for characters in the English language was the American Standard Code for Information Interchange (ASCII)

The encoding for each ASCII character fits in 1 byte (8 bits)

Built-in functions ord() and char()

```
>>> ord('a')
97
>>> ord('?')
63
>>> ord('\n')
10
>>> chr(10)
'\n'
>>> chr(63)
'?'
>>> chr(97)
'a'
>>>
```

Function `ord()` takes a character (i.e., a string of length 1) as input and returns its ASCII code

Function `chr()` takes an ASCII encoding (i.e., a non-negative integer) and returns the corresponding character

Beyond ASCII

A string object contains an ordered sequence of characters which can be any of the following:

- lowercase and uppercase letters in the English alphabet: `a b c … z` and `A B C … Z`
- decimal digits: `0 1 2 3 4 5 6 7 8 9`
- punctuation: `, . : ; ‘ “ ! ?` etc.
- Mathematical operators and common symbols: `= < > + - / * $ # % @ &` etc.
- Characters from languages other than English
- Technical symbols from math, science, engineering, etc.

There are only 128 characters in the ASCII encoding

Unicode has been developed to be the universal character encoding scheme

Unicode

In Unicode, every character is represented by an integer code point.

The code point is not necessarily the actual byte representation of the character; it is just the identifier for the particular character

The code point for letter a is the integer with hexadecimal value 0x0061

- Unicode conveniently uses a code point for ASCII characters that is equal to their ASCII code

With Unicode, we can write strings in

- english
- cyrillic
- chinese
- …

escape sequence \u indicates start of Unicode code point

```
>>> '\u0061'
'a'
>>> '\u0064\u0061d'
'dad'
>>>
'\u0409\u0443\u0431\u043e\u043c\u
0438\u0440'
'Љубомир'
>>> '\u4e16\u754c\u60a8\u597d!'
'世界您好!'
>>>
```

String comparison, revisited

Unicode code points, being integers, give a natural ordering to all the characters representable in Unicode

Unicode was designed so that, for any pair of characters from the same alphabet, the one that is earlier in the alphabet will have a smaller Unicode code point.

```
>>> s1 = '\u0021'
>>> s1
'!'
>>> s2 = '\u0409'
>>> s2
'Љ'
>>> s1 < s2
True
>>>
```

Unicode Transformation Format (UTF)

A Unicode string is a sequence of code points that are numbers from 0 to 0x10FFFF.

Unlike ASCII codes, Unicode code points are not what is stored in memory; the rule for translating a Unicode character or code point into a sequence of bytes is called an encoding.

There are several Unicode encodings: UTF-8, UTF-16, and UTF-32. UTF stands for Unicode Transformation Format.

- UTF-8 has become the preferred encoding for e-mail and web pages
- The default encoding when you write Python 3 programs is UTF-8.
- In UTF-8, every ASCII character has an encoding that is exactly the 8-bit ASCII encoding.

Assigning an encoding to “raw bytes”

When a file is downloaded from the web, it does not have an encoding

- the file could be a picture or an executable program, i.e. not a text file
- the downloaded file content is a sequence of bytes, i.e. of type `bytes`

The `bytes` method `decode()` takes an encoding description as input and returns a string that is obtained by applying the encoding to the sequence of bytes

- the default is UTF-8

```
>>> content
b'This is a text document\nposted on the\nWWW.\n'
>>> type(content)
<class 'bytes'>
>>> s = content.decode('utf-8')
>>> type(s)
<class 'str'>
>>> s
'This is a text document\nposted on the\nWWW.\n'
>>> s = content.decode()
>>> s
'This is a text document\nposted on the\nWWW.\n'
>>>
```