CptS 111, Spring 2023
Lect. #23, Apr. 17, 2023
Class Notes

Today's Agenda:

1. Checking for membership
2. Correcting a lie of omission
3. Some dictionary methods
4. Nested dictionaries

Ch. 9 (cont.)

Lists and Dictionaries (cont.)

1. Checking for Membership Using `in` in Lists

In Ch. 8, we discussed the use of `in` with strings, but it can be used with any container; in fact, you used `in` in PA #5 for a dictionary.

Let's consider a different kind of application. Suppose you have a long list of number, and you want to see whether it includes a particular number.

In [2]:
```
# Use of 'in' with list of integers (here, years)

years = [1066, 1812, 1776, 2019, 1492, 2001, 1993, 1991, 1960, 1923,
         1918, 1919, 1922, 2002, 1865, 1968, 2008, 1215, 1337, 1789,
         1848, 1815, 2011, 2004, 1994, 1957, 1969, 1945, 1929, 2008,
         1928, 1861, 1962, 1909, 1911, 1912, 1947, 1952, 1956, 1588,
         1665, 1666, 1348, 1337, 1859, 2002, 2003, 2004, 2005, 2022,
         2007, 2002, 2009, 2013, 1911, 1912, 1913, 1914, 1915, 1551,
         1509, 1661, 1671, 1672, 1673, 1771, 1779, 1917, 1941, 1944,
         1972, 1908, 1901, 1986, 2000, 1863, 1789, 1981, 1521, 1291,
         1413, 2023, 2020, 1963, 1950, 1989, 2050, 1865, 1517, 1792]
year = int(input('Enter year: '))
if year in years:
    print (f'Found {year} in list.')
else:
    print(f'{year} not in list.')
```

```
Enter year: 2001
Found 2001 in list.
```

Or you might have a long list of names and want to determine whether a particular name is in the list:

In [6]:
```
# Use of 'in' with list of strings

names = ['Smith', 'Wang', 'Delgado', 'Khaledian', 'Abnousi', 'Tao', 'Tran
         'Esna Ashari', 'Chowdhury', 'Clarke', 'Jones', 'Hawkes', 'Nguyen
         'Morris', 'Nand', 'Rowe', 'Booth', 'Warner', 'Davis-Early', 'Har
         'Zhang', 'Turner', 'Watt', 'de la Cruz', 'Reynolds', 'Raymer',
         'Chen', 'Sambisa', 'Bork', 'Lindauer', 'Obama', 'Schulz', 'Oriar
         'Hernandez', 'Callero', 'Higgins', 'Lockwood', 'Brayton', 'Tomso
         'Ramirez', 'Wen', 'Ma', 'De Goede', 'Ciot', 'Christian', 'Massey
         'Sharma', 'Weik', 'Waring', 'Al-Dalaan', 'Luquin', 'Kasper', 'Da
         'Eaton', 'Patel', 'Beck', 'Schneider', 'Peckham', 'Rodriguez',
         'Dutton', 'Griffin', 'Hinton', 'Foster', 'Larsen', 'Mock',
         'Jangthanasombat', 'Yoshimura', 'Birch', 'Stites', 'Ellovich', '
         'Pande', 'Azana', 'Tokuno', 'Chase', 'Sok', 'Casila', 'Newman']
name = input('Enter name: ')
if name in names:
    print('Found', name, 'in list.')
else:
    print(name, 'not in list.')
```

```
Enter name: Sambisa
Found Sambisa in list.
```

2. An Important Aside Regarding Tuples
(fixing a lie of omission)

Consider the following function which might look familiar to you.

In [7]:
```
# Returning multiple values from a function

def get_input():
    wt = float(input('Enter your weight in pounds: '))
    ht = float(input('Enter your height in inches: '))
    return wt, ht

def main():
    weight, height = get_input()
    print()
    print(f'Weight: {weight:g} pounds\nHeight: {height:g} inches')

main()
```

```
Enter your weight in pounds: 135
Enter your height in inches: 60

Weight: 135 pounds
Height: 60 inches
```

In the function above, `wt` and `ht` are both returned to the calling function. ***However, functions can really only return one object.*** In fact, Python interprets `wt, ht` as the tuple `(wt, ht)` and returns this tuple to the calling function or program. Python also interprets the lvalues `weight` and `height` as a single tuple, but it *unpacks* the tuple returned by the

function so you don't have to worry about tuples at all. Actually, it's probably safer to ignore the tupleness because it forces you to have the same number of lvalues on the left side as are in the tuple returned by the function.

In [8]:
```
# Python unpacks the tuple returned by the divmod() function

whole, remainder = divmod(11, 3)
print(whole, remainder)
```

```
3 2
```

In [9]:
```
# divmod() returns a tuple

result = divmod(11, 3)
print(result[0], result[1])
```

```
3 2
```

We haven't talked about the use of tuples a lot in this course, but we have used them a lot! It's just that often Python unpacked them for us. In fact, most of the time we've used values separated by commas, we've been using tuples. Let's look at two other instances of tuple use.

In [10]:
```
# No need for parentheses!

tuple_am_i = 1, 2, 3, 4
print(type(tuple_am_i))
print(tuple_am_i)
```

```
<class 'tuple'>
(1, 2, 3, 4)
```

In [11]:
```
# But use of parentheses doesn't guarantee tupleness

tuple_i_am_not = (42)
type(tuple_i_am_not)
```

Out[11]:
```
int
```

3. Common Dictionary Methods

In Ch. 3, we discussed the use of `del` and `.clear()` with dictionaries. `del` is used to delete one key-value pair, and `.clear()` is used to delete all key-value pairs. Today we'll cover a few more dictionary methods.

```
.get():    nonvoid method that returns value for a key or default
if key doesn't exist
.update(): void method that combines two dictionaries
.pop():    nonvoid method that returns & removes value for a key
or default if no key
```

In [12]:
```
# First set up two dictionaries

dict1 = dict(bananas=.69, satsumas=1.49, pomelos=1.29)
dict2 = dict(cherries=2.69, nectarines=1.69, satsumas=1.29)
print('dict1:', dict1)
print('dict2:', dict2)
```

```
dict1: {'bananas': 0.69, 'satsumas': 1.49, 'pomelos': 1.29}
dict2: {'cherries': 2.69, 'nectarines': 1.69, 'satsumas': 1.29}
```

In [13]:
```
# Note that operator overloading doesn't work with dictionaries!

dict3 = dict1 + dict2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [13], in <cell line: 3>()
      1 # Note that operator overloading doesn't work with dictionaries!
----> 3 dict3 = dict1 + dict2

TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

In [14]:
```
# Update dict1 using dict2; if key-value pair doesn't exist, it will be
# added; if it does, then value will be replaced from dict2

dict1.update(dict2)
print('dict1:', dict1)
print('dict2:', dict2)
```

```
dict1: {'bananas': 0.69, 'satsumas': 1.29, 'pomelos': 1.29, 'cherries': 2.69, 'nectarines': 1.69}
dict2: {'cherries': 2.69, 'nectarines': 1.69, 'satsumas': 1.29}
```

Compare the last three examples. Notice that `dict2` wasn't changed, but two entries were added to `dict1`, and one value was modified.

In [15]:
```
# Find value for key; return default if key doesn't exist

dict2.get('cherries', 'No such key')
```

Out[15]:
```
2.69
```

In [16]:
```
# Find value for key; return default if key doesn't exist

price = dict2.get('pineapple', 'No such key')
print('pineapple:', price)
```

```
pineapple: No such key
```

In [17]:
```
# Compare the previous example with the following for obtaining a value:

price = dict2['pineapple']  # Former way of obtaining a value using a key
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call l
ast)
Input In [17], in <cell line: 3>()
      1 # Compare the previous example with the following for obtaining
a value:
----> 3 price = dict2['pineapple']

KeyError: 'pineapple'
```

In [18]:
```
# Find value for key; return default if key doesn't exist; then delete
# key-value pair

print('dict1:', dict1)
price = dict1.pop('pineapple', 'No such key')
print('pineapple:', price)
```

```
dict1: {'bananas': 0.69, 'satsumas': 1.29, 'pomelos': 1.29, 'cherries':
2.69, 'nectarines': 1.69}
pineapple: No such key
```

In [19]:
```
# Compare this with delete

del dict1['pineapple'] # Former way of deleting a key-value pair
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call l
ast)
Input In [19], in <cell line: 3>()
      1 # Compare this with delete
----> 3 del dict1['pineapple']

KeyError: 'pineapple'
```

In [20]:
```
# Find value for key; return default if key doesn't exist; then delete
# key-value pair

print('dict1:', dict1)
price = dict1.pop('pomelos', 'No such key')
print('pomelos:', price)
```

```
dict1: {'bananas': 0.69, 'satsumas': 1.29, 'pomelos': 1.29, 'cherries':
2.69, 'nectarines': 1.69}
pomelos: 1.29
```

In [21]:
```
# We can see that the key-value pair deleted

print('dict1:', dict1)
```

```
dict1: {'bananas': 0.69, 'satsumas': 1.29, 'cherries': 2.69, 'nectarine
s': 1.69}
```

So finally, we can use the `.clear()` method to remove all the key-value pairs in a dictionary.

In [22]:
```
# Use .clear() to remove all key-value pairs:

dict1.clear()
print('dict1:', dict1)
```

```
dict1: {}
```

To summarize, we now have learned the following about dictionaries:

1. We know how to initialize an empty dictionary.
2. We know how to add key-value pairs to an empty dictionary.
3. We know several different ways (3!) to create a dictionary.
4. We know how to change the value of a key-value pair in a dictionary.
5. We know how to check for membership in a dictionary.
6. We know how to use a dictionary in an iterating `for` -loop using `dict_name.keys()`, `dict_name.values()`, or `dict_name.items()`.
7. We know how to use the void method `.update()` to update one dictionary with key-value pairs from another.
8. We know how to use the non-void method `.get()` to find the value (or a default value if a key-value pair doesn't exist) for a given key.
9. We know how to use the `del` command to delete a key-value pair, but we know a better way is to use the non-void `.pop()` method because it will return a default value if a key-value pair doesn't exist rather than an error.
10. Finally, we know how to use the void method `.clear()` to remove all key-value pairs from a dictionary.

4. Nested dictionaries

Next, let's turn to our last topic for dictionaries. Nested dictionaries can be very useful. Consider a dictionary called `grades` with three keys: `'Juan Dough'`, `'Jin Money'`, and `'Jae Moolah'`. The values of these keys are themselves dictionaries, each with the three keys: `'PAs'`, `'Midterms'`, and `'Final'`. The values of `'PAs'` and `'Midterms'` are lists; the value of `'Final'` is an integer.

In [23]:
```
# Create a nested dictionary with student names as keys and dictionaries
# values
grades = {
    'Juan Dough': {
        'PAs': [100, 88, 100],
        'Midterms': [85, 78],
        'Final': 90
    },
    'Jin Money': {
        'PAs': [92, 100, 100],
        'Midterms': [88, 94],
        'Final': 95
    },
    'Jae Moolah': {
        'PAs': [95, 97, 94],
        'Midterms': [91, 90],
        'Final': 89
    }
}
```

So how do we use this nested dictionary? Well, let's enter one of the outer keys as a name:

In [24]:
```
name = input('Enter student name: ')
```

```
Enter student name: Jin Money
```

If this name exists in the grades dictionary, we can find its value. In fact, because its value is a dictionary, we can use successive keys as additional values the way we use indexes for lists of lists. `grades[name]['PAs']` is the same as `grades['Juan Dough']['PAs']` which is the key to the list of PAs `[100, 88, 100]`. Let's see how this works:

In [25]:
```python
# First we check to see whether 'name' is in the dictionary 'grades'

if name in grades:
    print(f'Scores for {name}:')          # Print header

    # grades[name]['PAs'] is the key which returns a list value
    PAs = grades[name]['PAs']             # PAs is a list of grades

    # grades[name]['Midterms'] is the key which returns a list value
    midterms = grades[name]['Midterms']  # midterms is a list of grades

    # grades[name]['Final'] is the key which returns an integer value
    final = grades[name]['Final']         # final is an integer

    # Loops to print grades
    for pa, score in enumerate(PAs):      # Print PA scores
        print(f'PA {pa+1}:{score:>9}')

    for midterm, score in enumerate(midterms):    # Print mideterm scores
        print(f'Midterm {midterm+1}:{score:>4}')

    print(f'Final:{final:>8}')            # Print final score
```

```
Scores for Jin Money:
PA 1:       92
PA 2:      100
PA 3:      100
Midterm 1:  88
Midterm 2:  94
Final:      95
```

We can put this code into a loop and prompt for as many names as we want. Note the use of the `in` command in the conditional test.

In [26]:
```
# Use while-loop to print grades for as many students as desired

while True:
    name = input('Enter student name [return to exit]: ')
    print()
    if name == '':
        break
    if name in grades:
        print(f'Scores for {name}:')
        PAs = grades[name]['PAs']
        midterms = grades[name]['Midterms']
        final = grades[name]['Final']
        for pa, score in enumerate(PAs):
            print(f'PA {pa+1}:{score:>9}')
        for midterm, score in enumerate(midterms):
            print(f'Midterm {midterm+1}:{score:>4}')
        print(f'Final:{final:>8}')
        print()
```

```
Enter student name [return to exit]: Jin Money

Scores for Jin Money:
PA 1:       92
PA 2:      100
PA 3:      100
Midterm 1:  88
Midterm 2:  94
Final:      95

Enter student name [return to exit]: Jae Moolah

Scores for Jae Moolah:
PA 1:       95
PA 2:       97
PA 3:       94
Midterm 1:  91
Midterm 2:  90
Final:      89

Enter student name [return to exit]: Jae Money

Enter student name [return to exit]: Juan Dough

Scores for Juan Dough:
PA 1:      100
PA 2:       88
PA 3:      100
Midterm 1:  85
Midterm 2:  78
Final:      90

Enter student name [return to exit]:
```