COMP 204: Sets, Commenting & Exceptions

Yue Li
based on material from Mathieu Blanchette, Carlos Oliver Gonzalez and Christopher Cameron

February 3, 2019
Sets: the unordered container for unique things

- **Syntax:**
  
  ```python
  myset = {1, 2, 3} 
or
  myset = set([1, 2, 3]) (careful, myset = {} is an empty dictionary)
  ```

- Sets never contain duplicates. Python checks this using the `==` operator.

  ```
  >>> myset = set([1, 1, 2, 3])
  set([1, 2, 3]) #only keep unique values
  >>> myset.add(4)
  set([1, 2, 3, 4])
  >>> myset.add(1)
  set([1, 2, 3, 4])
  ```

- #get unique characters of string
  ```
  >>> charset = set("AAACCGGGA")
  {A, C, G}
  ```

- Sets can only contain immutable objects (like dictionary keys)
- Elements in sets do not preserve their order.
Useful set methods and operations

▶ Membership testing

```python
>>> 4 in myset
False
```

▶ Set intersection (elements common to A and B, if A and B are sets)

```python
>>> A = {"a", "b", "c"}
>>> B = {"a", "b", "d"}
>>> A & B  # equivalent to: A.intersection(B)
set(["a", "b"])
```

▶ Click [here](#) for a full list of set functionality.
Useful set methods and operations

▶ Set difference (elements in A that are not in B)

1 >>> A - B
2 set(["c"]) # same as: A.difference(B)

▶ Set union (Elements found in A or B)

1 >>> A | B # equivalent to: A.union(B)
2 set(["a", "b", "c", "d"])

▶ These can be applied to multiple sets

1 >>> C = {"a", "c", "d", "e"}
2 >>> A & B & C # A.intersection(B, C)
3 set(["a"]) # elements common to A and all others
Practice problems

1. Write a program that counts the number of unique letters in a given string. E.g. "bob" should give 2.

2. Write a program that checks whether a list of strings contains any duplicates. ['att', 'gga', 'att'] should return True.
# 1. long way
uniques = 0
for c in "bob":
    if c not in bob:
        uniques += 1

#1. short way
len(set("bob"))

#2. long way
uniques = []
mylist = ['att', 'gga', 'att']
for item in mylist:
    if item not in uniques:
        uniques.append('att')
if len(uniques) != len(mylist):
    print("found duplicates")

#3. short way
if len(set(mylist)) != len(mylist):
    print("found duplicates")
Practice problem: putting it all together

- You’re going to create your own dating app. Each user’s profile is a dictionary with the following keys:
  - 'movies' set of strings.
  - 'foods' set of strings.
  - 'genome' set of DNA strings.

- The user database will also be a dictionary where each key is a person’s name and the value is its profile dictionary.

- E.g. database['bob'] maps to

```python
{
    'movies': {'legally blonde', 'mission impossible'},
    'foods': {'mexican', 'vegetarian'},
    'genes': {'AAC', 'AAT', 'GGT', 'GGA'}
}
```
Your app will support 3 functions:

1. **add_user(name, profile, database)** creates a key for the user with its profile info and returns the updated database. (assume all names given are unique)

2. **compatibility_score(user_1, user_2, database)** Returns the compatibility score between two user profiles. Given as:
   
   ▶ similarity(u1, u2) = # of movies in common + # of foods in common + genome diversity i.e. number of genes in u1 or u2 but not in both.

3. **most_compatible(user, database)** returns user with the highest compatibility score to user.
Commenting: rules of thumb

- Comments should be informative but not overly detailed.
- Comments should be indented with the block they address

Which is better?

```
#this line binds an empty list to the name
'students'

students = []
for s in students:
  #loop over list and print
  print(s)
```

```
#keep track of students in a list
students = []
#display student list
for s in students:
  print(s)
```
Commenting: Docstrings

- A triple quoted string directly under a function header is stored as function documentation.

```python
def my_max(lili):
    
    """ Input: an iterable
    return: max of list
    """

    return max(lili)
```

```python
>>> help(my_max)
Help on function my_max in module __main__:

    my_max(lili)
    Input: an iterable
    return: max of list
```
Tips on coding style

▶ Be critical of your code. → is this the best it can be?
▶ Avoid hard-coding
  ▶ for i in range(len(mylist)) is better than
  ▶ for i in range(5)
▶ Give objects meaningful names. Avoid names like string, list, number, result, x, y
▶ When lines get too long you are either doing something wrong or you should break the line

```
for mylistitem in [innerlistitem in originallist if innerlistitem / 2 + 4 > 9]:
    print("hi")
```

▶ A complete description of Python’s coding style guidelines is here
Bugs: when things break

- You will probably have noticed by now that things don’t always go as expected when you try to run your code.
- We call this kind of occurrence a “bug”.
- One of the first uses of the term was in 1946 when Grace Hopper’s software wasn’t working due to an actual moth being stuck in her computer.

---

1 Wikipedia
Types of bugs

There are three major ways your code can go wrong.

1. Syntax errors
2. Exceptions (runtime)
3. Logical errors
Syntax Errors: “Furiously sleep ideas green colorless.”

► When you get a syntax error it means you violated a writing rule and the interpreter doesn’t know how to run your code.
► Your program will crash without running any other commands and produce the message `SyntaxError` with the offending line and a `^` pointing to the part in the line with the error.
► Game: spot the syntax errors!

```python
print("hello")
x = 0
while True
    x = x + 1
mylist = ["bob" 2, False]
if x < 1:
    print("x less than 1")
```

---

2 Noam Chomsky (1955)
Exceptions: “Colorless green ideas sleep furiously”³

▶ If you follow all the syntax rules, the interpreter will try to execute your code.

▶ However, the interpreter may run into code it doesn’t know how to handle so it raises an Exception

▶ The program has to deal with this Exception if it is not handled, execution aborts.

▶ Note: unlike with syntax errors, all the instructions before the interpreter reaches an exception do execute.

▶ Here is a list of all the built-in exceptions and some info on them.

³Noam Chomsky (1955)
Exceptions: ZeroDivisionError

- There are many types of exceptions, and eventually you will also be able to define your own exceptions.
- I'll show you some examples of common Exceptions.
- **ZeroDivisionError**

```python
x = 6
y = x / (x - 6)  # syntax is OK, executing fails

File "test.py", line 2, in <module>
y = x / (x - 6)
ZeroDivisionError: integer division or modulo by zero
```
Exceptions: NameError

- Raised when the interpreter cannot find a name-binding you are requesting.
- Usually happens when you forget to bind a name, or you are trying to access a name outside your namespace.

```python
def foo():
    x = "hello"
foo()
print(x)
```

```
Traceback (most recent call last):
  File "exceptions.py", line 4, in <module>
    print(x)
NameError: name 'x' is not defined
```
Exceptions: IndexError

- Raised when the interpreter tries to access a list index that does not exist

```python
mylist = ["bob", "alice", "nick"]
print(mylist[len(mylist)])
```

```
Traceback (most recent call last):
  File "exceptions.py", line 2, in <module>
    print(mylist[len(mylist)])
IndexError: list index out of range
```
Exceptions: TypeError

- Raised when the interpreter tries to do an operation on a non-compatible type.

```python
>>> mylist = ["bob", "alice", "nick"]
>>> mylist + "mary"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

```
# this is okay
>>> mylist * 2
["bob", "alice", "nick", "bob", "alice", "nick"]
```
When an exception is raised, you get a traceback message which tells you where the error was raised.

```python
def foo():
    return 5 / 0

def fee():
    return foo()

fee()
```

Traceback (most recent call last):
File "exception.py", line 5, in <module>
    fee()
  File "exception.py", line 4, in fee
    return foo()
  File "exception.py", line 2, in foo
    return 5 / 0
ZeroDivisionError: division by zero
Where do exceptions come from?

- Exceptions come from `raise` statements.
- **Syntax:** `raise [exception object]`
- You can choose to raise any exception object. Obviously a descriptive exception is preferred.
- You can even define your own exceptions but we leave this for a later lecture.

```python
def my_divide(a, b):
    if b == 0:
        raise ZeroDivisionError
    else:
        return a / b

def my_divide(a, b):
    if b == 0:
        raise TypeError
    else:
        return a / b
```
Handling Exceptions

▶ When an exception is raised, the exception is passed to the calling block.
▶ If the calling block does not handle the exception, the program terminates.

```python
#unhandled exception
def list_divide(numerators, denominators):
    ratio = []
    for a, b in zip(numerators, denominators):
        ratio.append(my_divide(a, b))
    return ratio
list_divide([1, 2, 1, 0], [1, 1, 0, 2])
```

Life Hack 1

The `zip(*args)` function lets you iterate over lists simultaneously. Yields tuple at each iteration with `(a[i], b[i])`.
try and except

- Python executes the **try** block.
- If the code inside the **try** raises an exception, python executes the **except** block.

```python
# exception handled by caller
def list_divide(numerators, denominators):
    ratio = []
    for a, b in zip(numerators, denominators):
        try:
            ratio.append(my_divide(a, b))
        except ZeroDivisionError:
            print("division by zero, skipping")
            continue
    return ratio

list_divide([1, 2, 1, 0], [1, 1, 0, 2])
```
Try/except: a more realistic example

- Often exceptions are caused by external users giving the program data it is not expecting.

```python
# not handling exceptions
while True:
    # if user gives invalid input program crashes
    x = int(input("Give me a number: "))

# handling exceptions
while True:
    try:
        x = int(input("Give me a number: "))
        break
    except TypeError:
        print("Not a number! Try again.")
```
Try/except/else: when no exception occurs

- An `else` block after a try/catch executes **only** if the `try` does not cause an exception.

```python
while True:
    try:
        a = int(input("Give me a numerator: "))
        b = int(input("Give me a denominator: "))
    except:
        print("Not a number! Try again."")
    else:
        print(f"{a} divided by {b} is {my_divide(a, b)}")
    break
```
while True:
    try:
        a = int(input("Give me a numerator: "))
        b = int(input("Give me a denominator: "))
        print(f"{a} divided by {b} is {my_divide(a, b)}")
        break
    except:
        print("Not a number! Try again.")
And finally, the **finally** statement

- The **finally** block **always** executes after the **try**-**except**-**else** blocks.

- Useful when:
  1. The **except** or **else** block itself throws an exception.
  2. The **try** throws an unexpected exception.
  3. A control flow statement in the **except** skips the rest.

- Why is it useful? Often there are statements you need to perform before your program closes. If there is an exception you forgot to handle, the finally will still execute.
```python
while True:
    try:
        a = int(input("Give me a numerator: "))
        b = int(input("Give me a denominator: "))
    except:
        print("Not a number! Try again.")
        break
    else:
        result = my_divide(a, b)
    finally:
        print("hello from finally!")
        print("hello from the other siiiiide")
```
Okay one last thing: **assert**

- The **assert** statement is a shortcut to raising exceptions.
- Sometimes you don’t want to execute the rest of your code unless some condition is true.

```python
def divide(a, b):
    assert b != 0
    return a / b
```

- If the **assert** evaluates to True then an **AssertionError** exception is raised.
- **Pro**: quick and easy to write
- **Con**: exception error may not be so informative.
- Used mostly for debugging and internal checks than for user friendliness.
Logical errors

- When according to Python your code is fine and runs without errors but it does not do what you intended.

- Example: spot the logical error

```python
#1
def my_max(mylist):
    for bla in mylist:
        my_max = 0
        if bla > my_max:
            my_max = bla
    return my_max
```

- There’s nothing to do to avoid logical errors other than testing your code thoroughly and having a good algorithm.

- Logical errors are often silent but **deadly**.