COMP 204

Exceptions (continued) and Sets

Mathieu Blanchette
based on material from Yue Li, Carlos Oliver Gonzalez and Christopher Cameron
Traceback (exceptions can be caused by user input)

def BMI(weight, height):
    print("Computing BMI")
    bmi = weight / (height * height)
    print("Done computing BMI")
    return bmi

def get_BMI_from_user():
    w = int(input("Please enter weight "))
    h = int(input("Please enter height "))
    bmi = BMI(w, h)
    return bmi

myBMI = get_BMI_from_user()

# Output:
# Please enter weight 4
# Please enter height 0
# Computing BMI
# Traceback (most recent call last):
# File "excTraceBack.py", line 13, in <module>
#     myBMI = get_BMI_from_user()
# File "excTraceBack.py", line 10, in <module>
#     bmi = BMI(w, h)
# File "excTraceBack.py", line 3, in <module>
#     return weight / (height * height)
# builtins.ZeroDivisionError: division by zero
When Exceptions is not handled

- If a function generates an Exception but does not handle it, the Exception is sent back to the calling block.
- If the calling block does not handle the exception, the Exception is sent back to its calling block... etc.
- If no-one handles the Exception, the program terminates and reports the Exception.

```
get_BMI_from_user() → function call → BMI(w,h) → function call → bmi = weight/(height*height)
```

ZeroDivisionError exception ZeroDivisionError exception
Handling Exceptions: try and except

A program can provide code to handle an Exception, so that it doesn’t crash when one happens.

- To be able to handle an exception generated by a piece of code, that code needs to be within a try block.
- If the code inside the try block raises an exception, its execution stops and the interpreter looks for code to handle the Exception.
- Code for handling Exception is in the except block.

```python
try:
    # do something that may cause an Exception
    # some more code
except <SomeExceptionType>:
    # do something to handle the Exception
    # rest of code
```

If L2 raises an Exception of type SomExceptionType, we jump to L4, without executing L3.

If L2 doesn’t cause an exception, L3 is executed, and L4 and 5 are not executed.
BMI function handles the Exceptions it caused.

```python
def BMI(weight, height):
    print("Computing BMI")
    try:
        bmi = weight / (height * height)
        print("Done computing BMI")
    except ZeroDivisionError:
        print("There was a division by zero")
        bmi = -1  # a special code to indicate an error
    return bmi

def get_BMI_from_user():
    w = int(input("Please enter weight "))
    h = int(input("Please enter height "))
    bmi = BMI(w, h)
    print("Thank you!")
    return bmi

myBMI = get_BMI_from_user()
# Please enter weight 4
# Please enter height 0
# Computing BMI
# There was a division by zero
# Thank you!
```
BMI function does not handle the Exceptions is causes.
get_BMI_from_user handles the Exception raised in BMI function.

```python
def BMI(weight, height):
    print("Computing BMI")
    bmi = weight / (height * height)
    print("Done computing BMI")
    return bmi

def get_BMI_from_user():
    w = int(input("Please enter weight "))
    h = int(input("Please enter height "))
    try:
        bmi = BMI(w, h)
        print("Thank you!")
    except:
        print("There was a problem computing BMI")
bmi=-1
    return bmi

myBMI = get_BMI_from_user()
# Please enter weight 4
# Please enter height 0
# Computing BMI
# There was a problem computing BMI
```
Raising our own Exceptions

- Exceptions come from `raise` statements.

- Syntax: `raise [exception object]`

- You can choose to raise any exception object. Obviously a descriptive exception is preferred.

```python
def my_divide(a, b):
    if b == 0:
        raise ZeroDivisionError
    else:
        return a / b
```
We can raise an informative exception

```python
# This BMI function raises a ValueError Exception if the weight or height are <= 0

def BMI(weight, height):
    if weight <= 0 or height <= 0:
        raise ValueError("BMI handles only positive values")
    print("Computing BMI")
    return weight / (height * height)

def get_BMI_from_user():
    w = int(input("Please enter weight "))
    h = int(input("Please enter height "))
    bmi = BMI(w, h)
    print("Thank you!")
    return bmi

myBMI = get_BMI_from_user()

# Traceback (most recent call last):
# File "excTraceBack.py", line 16, in <module>
#   myFunction()
# File "excTraceBack.py", line 12, in <module>
#   r = ratio(5,0)
# File "excTraceBack.py", line 5, in <module>
#   raise ValueError("BMI handles only positive values")
# builtins.ValueError: BMI handles only positive values
```
Handling exceptions raised from one function in another

```python
# This BMI function raises a ValueError Exception
# if the weight or height are <= 0

def BMI(weight, height):
    if weight <= 0 or height <= 0:
        raise ValueError("BMI handles only positive values")
    print("Computing BMI")
    return weight / (height * height)

def get_BMI_from_user():
    while True:  # keep asking until valid entry is obtained
        w = int(input("Please enter weight "))
        h = int(input("Please enter height "))
        try:
            bmi = BMI(w, h)
            print("Thank you!")
            break  # stop asking, break out of the loop
        except ValueError:
            print("Error calculating BMI")

    return bmi

myBMI = get_BMI_from_user()```
What if user enters a string that cannot be converted to an integer? (e.g. "Twelve")

This would cause a ValueError Exception within the int() function.

To be more robust, our program should catch that Exception and deal with it properly.
```python
def BMI(weight, height):
    if weight <= 0 or height <= 0:
        raise ValueError("BMI handles only positive values")
    print("Computing BMI")
    return weight / (height * height)

def get_BMI_from_user():
    while True:  # keep asking until valid entry is obtained
        try:
            w = int(input("Please enter weight "))
            h = int(input("Please enter height "))
        except ValueError:  # exception raised from int()
            print("Please only enter integers")
        else:
            try:
                bmi = BMI(w, h)
                print("Thank you!")
                break  # stop asking, break out of the loop
            except ValueError:  # exception raised from BMI()
                print("Error calculating BMI")
    return bmi

myBMI = get_BMI_from_user()

Note: Use `else` block after a try/catch executes **only** if the `try` does not cause an exception.
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 False 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.
Sets

October 18, 2019
Sets: the unordered container for unique things

A set is a compound type (like Lists, Tuples, Strings, Dictionaries)
  ▶ Stores an unordered set of objects (no indexing possible)
  ▶ No duplicates
  ▶ Can contain only immutable objects

A Set offers a limited version of the functionality of a List, which enables it to perform its operations faster.
Sets: the unordered container for unique things

- **Syntax:** `myset = {1, 2, 3}` (careful, `myset = {}` is an empty dictionary)
- We can create a set from a list: `myset = set([1, 2, 3])` or `myset = set([])`
- We can create a set from a string: `myset = set("ACGAA")`  
  ## `myset is {A, C, G}`
- Sets never contain duplicates. Python checks this using the `==` operator.
- To add an element to a set, use the add function:

```python
>>> 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])
```
Useful set methods and operations

Click here for a full list of set functionality.

- **Number of elements:** `len(myset)` ## 4
- **Membership testing:** `if 5 in myset:` ## False
- **Iterating through set:** `for element in myset:

- **Set intersection (elements common to A and B)**

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

▶ Set union (Elements found in A or B)

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

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

```
>>> A - B
set(['c'])  #same as: A.difference(B)
```

▶ These can be applied to multiple sets

```
>>> C = {'a', 'c', 'd', 'e'}

>>> A & B & C
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 = []
for c in "bob":
    if c not in uniques:
        uniques.append(c)

len(uniques)

# 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")