COMP 204

Exceptions (continued) and Sets

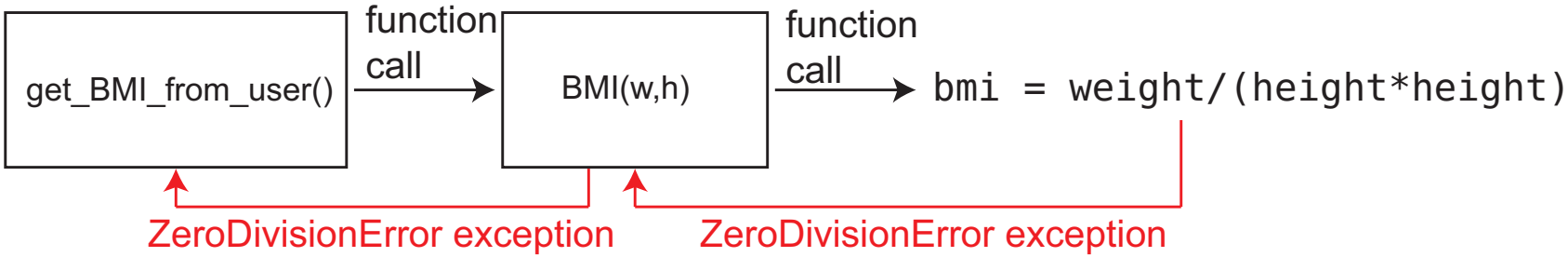
Mathieu Blanchette
based on material from Yue Li, Carlos Oliver Gonzalez and Christopher Cameron

Traceback (exceptions can be caused by user input)

```
1  def BMI(weight, height):
2      print("Computing BMI")
3      bmi = weight / (height * height)
4      print("Done computing BMI")
5      return bmi
6
7  def get_BMI_from_user():
8      w = int(input("Please enter weight "))
9      h = int(input("Please enter height "))
10     bmi = BMI(w,h)
11     return bmi
12
13 myBMI = get_BMI_from_user()
14 # Output:
15 # Please enter weight 4
16 # Please enter height 0
17 # Computing BMI
18 # Traceback (most recent call last):
19 #   File "excTraceBack.py", line 13, in <module>
20 #     myBMI = get_BMI_from_user()
21 #   File "excTraceBack.py", line 10, in <module>
22 #     bmi = BMI(w,h)
23 #   File "excTraceBack.py", line 3, in <module>
24 #     return weight / (height * height)
25 # builtins.ZeroDivisionError: division by zero
```

When Exceptions is not handled

- If a function generates an Exception but does not handle it, the Exception is send 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.



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.

```
1 try:
2     # do something that may cause an Exception
3     # some more code
4 except <SomeExceptionType>:
5     # do something to handle the Exception
6 # 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.

```
1  def BMI(weight, height):
2      print("Computing BMI")
3      try:
4          bmi = weight / (height * height)
5          print("Done computing BMI")
6      except ZeroDivisionError:
7          print("There was a division by zero")
8          bmi = -1  # a special code to indicate an error
9      return bmi
10
11 def get_BMI_from_user():
12     w = int(input("Please enter weight "))
13     h = int(input("Please enter height "))
14     bmi = BMI(w,h)
15     print("Thank you!")
16     return bmi
17
18 myBMI = get_BMI_from_user()
19 # Please enter weight 4
20 # Please enter height 0
21 # Computing BMI
22 # There was a division by zero
23 # Thank you!
```

BMI function does not handle the Exceptions is causes.
get_BMI_from_user handles the Exception raised in BMI function.

```
1  def BMI(weight, height):
2      print("Computing BMI")
3      bmi = weight / (height * height)
4      print("Done computing BMI")
5      return bmi
6
7  def get_BMI_from_user():
8      w = int(input("Please enter weight "))
9      h = int(input("Please enter height "))
10     try:
11         bmi = BMI(w,h)
12         print("Thank you!")
13     except:
14         print("There was a problem computing BMI")
15         bmi=-1
16     return bmi
17
18 myBMI = get_BMI_from_user()
19 # Please enter weight 4
20 # Please enter height 0
21 # Computing BMI
22 # 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.

```
1 def my_divide(a, b):
2     if b == 0:
3         raise ZeroDivisionError
4     else:
5         return a / b
```

We can raise an informative exception

```
# 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

```
1  # This BMI function raises a ValueError Exception
2  # if the weight or height are <= 0
3  def BMI(weight, height):
4      if weight <=0 or height <= 0 :
5          raise ValueError("BMI handles only positive values")
6      print("Computing BMI")
7      return weight / (height * height)
8
9  def get_BMI_from_user():
10     while True:  # keep asking until valid entry is obtained
11         w = int(input("Please enter weight "))
12         h = int(input("Please enter height "))
13         try:
14             bmi = BMI(w,h)
15             print("Thank you!")
16             break  # stop asking, break out of the loop
17         except ValueError:
18             print("Error calculating BMI")
19
20     return bmi
21
22 myBMI = get_BMI_from_user()
```

How to handle invalid user inputs by `try ... except`

- 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.

```
1  def BMI(weight, height):
2      if weight <=0 or height <= 0 :
3          raise ValueError("BMI handles only positive values")
4      print("Computing BMI")
5      return weight / (height * height)
6
7  def get_BMI_from_user():
8      while True:  # keep asking until valid entry is obtained
9          try:
10             w = int(input("Please enter weight "))
11             h = int(input("Please enter height "))
12         except ValueError: # exception raised from int()
13             print("Please only enter integers")
14         else:
15             try:
16                 bmi = BMI(w,h)
17                 print("Thank you!")
18                 break  # stop asking, break out of the loop
19             except ValueError: # excepion raised from BMI()
20                 print("Error calculating BMI")
21     return bmi
22
23 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.

```
1 def divide(a, b):
2     assert b != 0
3     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:

```
1 >>> myset = set([1, 1, 2, 3])
2 set([1, 2 , 3]) #only keep unique values
3 >>> myset.add(4)
4 set([1, 2, 3, 4])
5 >>> myset.add(1)
6 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)

```
>>> 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`

```python
# 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")
```