COMP 204
Exceptions and Miscellaneous

Mathieu Blanchette,
based on material from Carlos Oliver Gonzalez and Christopher Cameron
CSUS is holding a midterm review session on **Monday, October 15th, from 6-9pm**. The Facebook group is here: https://www.facebook.com/events/1721064144671158/
Exceptions: “Colorless green ideas sleep furiously”

- 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.

---

1Noam Chomsky (1955)
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.

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

# Output:
# File "excDivZero.py", line 2, in <module>
# y = x / (x - 6)
# builtins.ZeroDivisionError: division by zero
```
Exceptions: NameError

- Raised when the interpreter cannot find a variable you are requesting.
- Usually happens when you forget to create a name, or you are trying to access a variable outside its scope.

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

foo()
print(x)
```

# Output:

```
# Traceback (most recent call last):
# File "excName.py", line 5, in <module>
#   print(x)
#   builtins.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)])

# Output:

# Traceback (most recent call last):
# File "exIndex.py", line 2, in <module>
#     print(mylist[len(mylist)])
# builtins.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"]
newList = mylist + "mary"

# Output
# Traceback (most recent call last):
# File "excType.py", line 2, in <module>
#   newList = mylist + "mary"
# builtins.TypeError: can only concatenate list (not "str") to list
```
Traceback

What happens when an Exception is raised? The program’s normal control flow is altered.

- The execution of the block of code stops
- Python looks for code to handle the Exception (try/except block; see later)
- If it doesn’t find that code, it stops the program and produces a traceback message that tells you where the error was raised, which function it sits in, what code called that function, etc.
- See example on next slide...
```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 "))
    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
```
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.

```
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 L5, without executing L4.
If L2 doesn’t cause an exception, L4 is executed, and L5/6 are not executed.
In both cases, the program continues executing with L8.
```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)
    return bmi

myBMI = get_BMI_from_user()

# Output:
# Please enter weight 4
# Please enter height 0
# Computing BMI
# There was a division by zero
# Thank you!
```
Where do exceptions come from?

- Exceptions come from `raise` statements.
- The Python interpreter raises exception when it encounters errors.
- The programmer can also raise Exceptions!
- Syntax: `raise [exception object]`
# 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: try and except

- 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 send back to its calling block... etc.
- If no-one handles the Exception, the program terminates and reports the 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():
    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!")
        except ValueError:
            print("Error calculating BMI")
        break  # stop asking, break out of the loop
    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.
# 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
        try:
            w = int(input("Please enter weight "))
            h = int(input("Please enter height "))
        except ValueError:
            print("Please only enter integers")
            continue  # don't calculate BMI
        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()
There exist many ways to format strings for printing. **Formatted String Literals** are very useful:

```python
pi = 3.1415927

# standard printing
print('pi is', pi)

# printing using formatted strings
print(f'pi is {pi}')
print(f'pi is approx. {pi:.3f}')  # to round to 3 decimals

grades = {'Sjoerd': 8, 'Jack': 74, 'Annie': 100}
for name, grade in grades.items():
    # prints name over 10 characters, and grade over 5
    print(f'{name:10} ===> {grade:5d}')
```

#output:
# pi is 3.1415927
# pi is 3.1415927
# pi is approx. 3.142
# Sjoerd ===> 8
# Jack ===> 74
# Annie ===> 100

See Section 7.1 of https://docs.python.org/3/tutorial/inputoutput.html
Often, we need to iterate over the elements of two lists in parallel. The **zip** function allows us to do this easily.

**Example:** Assemble list of full names from list of first names and list of last names

```python
# without the zip function, assembling full names
# is a bit complicated
fullNames = []
for index in range(0, len(firstNames)):
    fullNames.append(firstNames[index] + " " + lastNames[index])
print(fullNames)

# or
fullNames = []
for index, first in enumerate(firstNames):
    fullNames.append(first + " " + lastNames[index])
print(fullNames)

# This is easier to do with the zip function
fullNames = []
for first, last in zip(firstNames, lastNames):
    fullNames.append(first + " " + last)
```

```
#output:
# [ 'Amol Prakash', 'Ahmed ElKhoury', 'Ayana Jones']
```