COMP 204
Exceptions

Mathieu Blanchette
based on material from Yue Li, Carlos Oliver Gonzalez and Christopher Cameron
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 (semantic) 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")
```

²Noam 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 encounter into code it is unable to execute, 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.

\(^3\)Noam 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
going to 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
g<<< 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"]
```
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...
Traceback

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
```
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.

```
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.
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():
    try:
        w = int(input("Please enter weight "))
        h = int(input("Please enter height "))
    except:
        print("invalid inputs")
        return 0
    bmi = BMI(w, h)
    print("Thank you!")
    return bmi

myBMI = get_BMI_from_user()

# Output:
# Please enter weight 4
# Please enter height 0
# Computing BMI
# There was a division by zero
Where do exceptions come from? We raise them

- 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 (out of scope).

```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  # we can raise any exception we want
    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 ** 2)

def get_BMI_from_user():
    while True:
        w = int(input("Please enter weight "))
        h = int(input("Please enter height "))
        try:
            bmi = BMI(w, h)
            print("Thank you!")
            break
        except ValueError:
            print("Error calculating BMI")

    return bmi

myBMI = get_BMI_from_user()
```

How to handle invalid user inputs by using `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.
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")
                continue  # don't calculate BMI, re-iterate
        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()
```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:
        try:
            w = int(input("Please enter weight "))
            h = int(input("Please enter height "))
        except ValueError:
            print("Please only enter integers")
        else:
            try:
                bmi = BMI(w, h)
                print("Thank you!")
                break
            except ValueError:
                print("Error calculating BMI")
    return bmi

myBMI = get_BMI_from_user()
```
Chained `except`

- Use `except` to catch different exceptions.
- Use `else` block after a `try/catch` executes **only** if the `try` does not cause an exception.

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

while True:
    try:
        a = int(input("Give me a numerator: "))
        b = int(input("Give me a denominator: "))
        result = my_divide(a, b)
    except ValueError:
        print("Not a number")
    except ZeroDivisionError:
        print("Can't divide by zero")
    else:
        print(f"{a} divided by {b} is {result}")
    break
```
Side-track: a convenient way to format print (Misc.)

There exist many ways to format strings for printing (Section 7.1). Formatted String Literals are very useful:

```python
import math
# standard printing
print('pi is', math.pi)

# printing using formatted strings
print(f'pi is {math.pi}')
print(f'pi is approx. {math.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 3.141592653589793
# pi is 3.141592653589793
# pi is approx. 3.142
# Sjoerd ==> 8
# Jack ==> 74
# Annie ==> 100
```
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:
                    "))
        result = my_divide(a, b)
    except ValueError:
        print("Not a number! Try again.\")
    except ZeroDivisionError:
        print("Can't divide by zero")
    else:
        print(f"{a} divided by {b} is {result}")
    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
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.
Often, we need to iterate over the elements of two lists in parallel

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

def list_divide(numerators, denominators):
    ratio = []
    for a, b in zip(numerators, denominators):
        print(f"dividing {a} by {b}\"")
        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]])
```
More examples on `zip` function (misc)

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

1. `firstNames = ['Amol', 'Ahmed', 'Ayana']`
2. `lastNames = ['Prakash', 'ElKhoury', 'Jones']`
3. # without the `zip` function, assembling full names
   # is a bit complicated
4. `fullNames = []`
5. `for index in range(0, len(firstNames)):`
6.    `fullNames.append(firstNames[index] + " " + lastNames[index])`
7. `print(fullNames)`
8. # or
9. `fullNames = []`
10. `for index, first in enumerate(firstNames):`
11.    `fullNames.append(first + " " + lastNames[index])`
12. `print(fullNames)`
13. # This is easier to do with the `zip` function
14. `fullNames = []`
15. `for first, last in zip(firstNames, lastNames):`
16.    `fullNames.append(first + " " + last)`
17. `print(fullNames)`
18. # output:
19. # ['Amol Prakash', 'Ahmed ElKhoury', 'Ayana Jones']
Types of bugs

1. Syntax errors
2. Exceptions (runtime)
3. Logical errors
Last type of bug: 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
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.