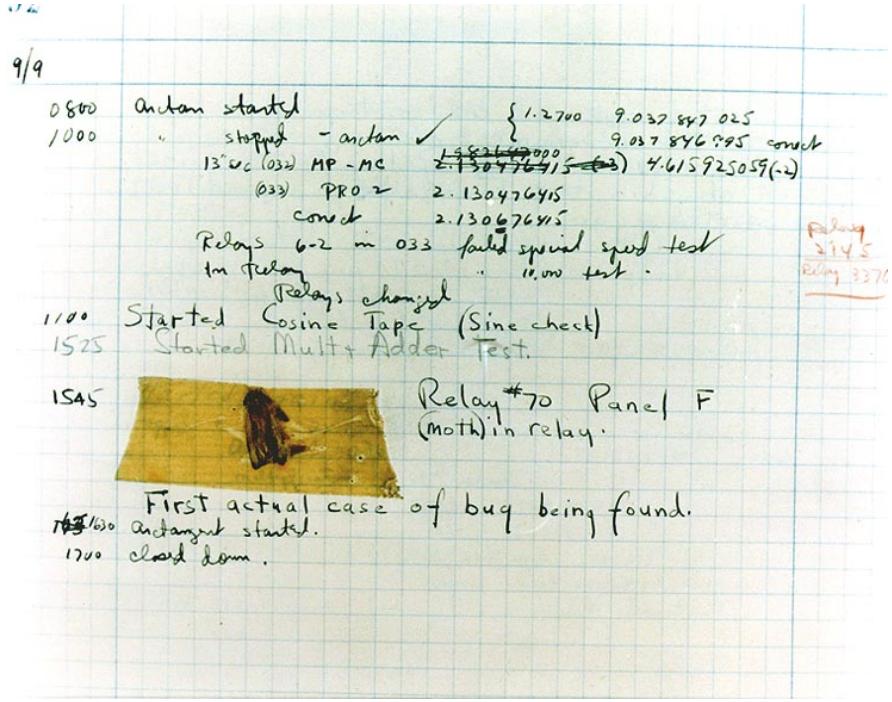
COMP 204

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]

[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."[2]

- 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!

```
1 print("hello)
2 x = 0
3 while True
4     x = x + 1
5 mylist = ["bob" 2, False]
6 if x < 1:
7 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 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`

```
1  x = 6
2  y = x  / (x - 6) #syntax is OK, executing fails
3
4  File "test.py", line 2, in <module>
5  y = x / (x - 6)
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.

```
1  def foo():
2      x = "hello"
3  foo()
4  print(x)
5  Traceback (most recent call last):
6    File "exceptions.py", line 4, in <module>
7      print(x)
8  NameError: name 'x' is not defined
```

Exceptions: IndexError

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

```
1 mylist = ["bob", "alice", "nick"]
2 print(mylist[len(mylist)])
3
4 Traceback (most recent call last):
5   File "exceptions.py", line 2, in <module>
6     print(mylist[len(mylist)])
7 IndexError: list index out of range
```

Exceptions: TypeError

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

```
>>> 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"]
```

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...

Traceback

- When an exception is raised, you get a traceback message which tells you where the error was raised.

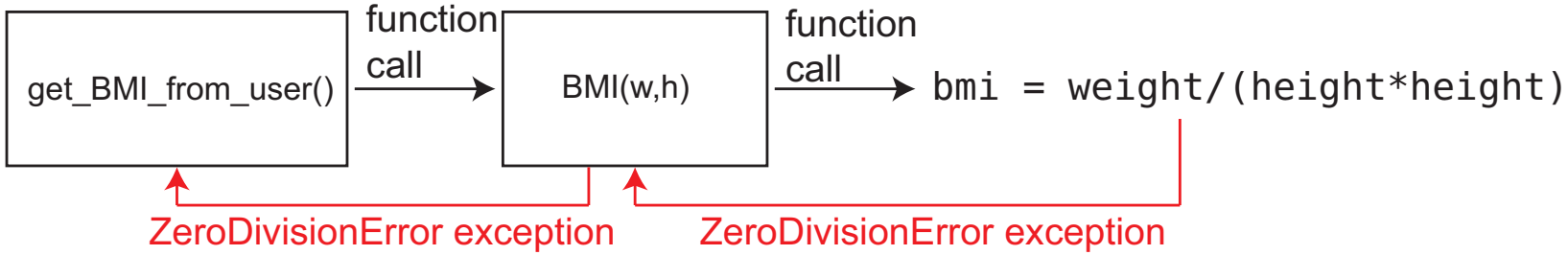
```
    def foo():
        return 5 / 0
    def fee():
        return foo()
    fee()

    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.



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.

```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():
    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).

```
1  def my_divide(a, b):
2      if b == 0:
3          raise ZeroDivisionError
4      else:
5          return a / b
6  def my_divide(a, b):
7      if b == 0:
8          raise TypeError # we can raise any exception
               ↪  we want
9      else:
10         return a / b
```

We can raise an informative exception

```
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     w = int(input("Please enter weight "))
11     h = int(input("Please enter height "))
12     bmi = BMI(w,h)
13     print("Thank you!")
14     return bmi
15
16 myBMI = get_BMI_from_user()
17 __
18 # Traceback (most recent call last):
19 #   File "excTraceBack.py", line 16, in <module>
20 #     myFunction()
21 #   File "excTraceBack.py", line 12, in <module>
22 #     r = ratio(5,0)
23 #   File "excTraceBack.py", line 5, in <module>
24 #     raise ValueError("BMI handles only positive values")
25 # 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.

Catch exceptions from `int()` and `continue`

```
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             continue  # don't calculate BMI, re-iterate
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
22     return bmi
23
24 myBMI = get_BMI_from_user()
```

try, except, else

```
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()
```

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.

```
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 denomenator: "))
        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:

```
1  import math
2  # standard printing
3  print('pi is',math.pi)
4
5  # printing using formatted strings
6  print(f'pi is {math.pi}')
7  print(f'pi is approx. {math.pi:.3f}') # to round to 3
       decimals
8
9  grades = {'Sjoerd': 8, 'Jack': 74, 'Annie': 100}
10 for name, grade in grades.items():
11     # prints name over 10 characters, and grade over 5
12     print(f'{name:10} ==> {grade:5d}')
13
14 #output:
15 # pi is 3.141592653589793
16 # pi is 3.141592653589793
17 # pi is approx. 3.142
18 # Sjoerd     ==>     8
19 # Jack       ==>    74
20 # 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` trows 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.

`finally` example

```
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.

```
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.

Misc: `zip` function

Often, we need to iterate over the elements of two lists in parallel

```
1  #unhandled exception
2  def list_divide(numerators, denominators):
3      ratio = []
4      for a, b in zip(numerators, denominators):
5          ratio.append(my_divide(a, b))
6      return ratio
7  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 $(a[i], b[i])$.

`zip` example with `try`, `except`, `continue`

```
1  def my_divide(a, b):
2      if b == 0:
3          raise ZeroDivisionError
4      else:
5          return a/b
6  def list_divide(numerators, denomenators):
7      ratio=[]
8      for a,b in zip(numerators, denomenators):
9          print(f"dividing {a} by {b}")
10         try:
11             ratio.append(my_divide(a,b))
12         except ZeroDivisionError:
13             print("division by zero, skipping")
14             continue
15     return ratio
16
17 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

```
firstNames = ['Amol','Ahmed','Ayana']
lastNames = ['Prakash','ElKhoury','Jones']
# 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)
print(fullNames)
#output:
# ['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

```
1  def my_max(mylist):
2      for bla in mylist:
3          my_max = 0
4          if bla > my_max:
5              my_max = bla
6      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.