

Principles of Computer Science II

Errors & Abstract Data Types

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 13



Syntax Errors

Until now error messages have not been more than mentioned. There are (at least) two distinguishable kinds of errors:

- ▶ syntax errors and
- ▶ exceptions.

```
1>>> while True print('Hello world')
2   File "<stdin>", line 1
3     while True print('Hello world')
4         ^
5 SyntaxError: invalid syntax
```

- ▶ File name and line number are printed so you know where to look in case the input came from a script.



Exceptions

1. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
2. Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs.
3. Most exceptions are not handled by programs, however, and result in error messages.

```
1>>> 10 * (1/0)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 ZeroDivisionError: division by zero
```



Exceptions: Examples

```
1>>> 4 + spam*3
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 NameError: name 'spam' is not defined
5
6>>> '2' + 2
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 TypeError: Can't convert 'int' object to str implicitly
```

- ▶ The last line of the error message indicates what happened.
- ▶ Exceptions come in different types, and the type is printed as part of the message.
- ▶ Standard exception names are built-in identifiers (not reserved keywords).
- ▶ We are allowed to define our own exceptions.



Handling Exceptions

- ▶ It is possible to write programs that handle selected exceptions.

```
1 while True:
2     try:
3         x = int(input("Please enter a number: "))
4         break
5     except ValueError:
6         print("Oops! That was no valid number. Try again
          ...")
```



Try statement

The try statement works as follows:

1. First, the **try** clause (the statement(s) between the **try** and **except** keywords) is executed.
2. If no exception occurs, the **except** clause is skipped and execution of the **try** statement is finished.
3. If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the **except** clause is executed, and then execution continues after the **try** statement.
4. If an exception occurs which does not match the exception named in the **except** clause, it is passed on to outer **try** statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.



Try statement

- ▶ A try statement may have more than one except clause, to specify handlers for different exceptions.
- ▶ At most one handler will be executed.
- ▶ Handlers only handle exceptions that occur in the corresponding **try** clause, not in other handlers of the same try statement.

```
1 except (RuntimeError, TypeError, NameError):
2     pass
```



Last Try statement

- ▶ The last except clause may omit the exception name(s), to serve as a wildcard.

```
1 import sys
2
3 try:
4     f = open('myfile.txt')
5     s = f.readline()
6     i = int(s.strip())
7 except OSError as err:
8     print("OS error: {0}".format(err))
9 except ValueError:
10    print("Could not convert data to an integer.")
11 except:
12    print("Unexpected error:", sys.exc_info()[0])
13    raise
```



Else Statement

- ▶ The `try ... except` statement has an optional else clause, which, when present, must follow all except clauses. It is useful for code that must be executed if the `try` clause does not raise an exception.

```
1 for arg in sys.argv[1:]:
2     try:
3         f = open(arg, 'r')
4     except OSError:
5         print('cannot open', arg)
6     else:
7         print(arg, 'has', len(f.readlines()), 'lines ')
8         f.close()
```



Exception details

- ▶ When an exception occurs, it may have an associated value, also known as the exceptions argument.
- ▶ The presence and type of the argument depend on the exception type.

```
1 try:
2     raise Exception('spam', 'eggs ')
3 except Exception as inst:
4     print(type(inst))    # the exception instance
5     print(inst.args)    # arguments stored in .args
6     print(inst)         # __str__ allows args to be
                          # printed directly,
                          # but may be overridden in
                          # exception subclasses
7
8     x, y = inst.args    # unpack args
9     print('x =', x)
10    print('y =', y)
```



Raising Exceptions

- ▶ The `raise` statement allows the programmer to force a specified exception to occur.
- ▶ The sole argument to `raise` indicates the exception to be raised.

```
1>>> raise NameError('HiThere')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 NameError: HiThere
```



User Defined Exceptions

- ▶ Programs may name their own exceptions by creating a new exception class.
- ▶ Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

```
1 class Error(Exception):
2     """Base class for exceptions in this module."""
3     pass
```



User Defined Exceptions: An Example

```
1 class InputError(Error):
2     """ Exception raised for errors in the input.
3
4     Attributes:
5         expression — input expression in which the error
6             occurred
7         message — explanation of the error
8     """
9     def __init__(self, expression, message):
10        self.expression = expression
11        self.message = message
```



User Defined Exceptions: An Example

```
1 class TransitionError(Error):
2     """ Raised when an operation attempts a state transition
3         that's not
4         allowed.
5
6     Attributes:
7         previous — state at beginning of transition
8         next — attempted new state
9         message — explanation of why the specific
10            transition is not allowed
11
12     """
13
14     def __init__(self, previous, next, message):
15        self.previous = previous
16        self.next = next
17        self.message = message
```



Clean Up Actions

- ▶ The **try** statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances.

```
1 try:
2     raise KeyboardInterrupt
3 finally:
4     print('Goodbye, world!')
```



Clean Up Actions: An Example

```
1 def divide(x, y):
2     try:
3         result = x / y
4     except ZeroDivisionError:
5         print("division by zero!")
6     else:
7         print("result is", result)
8     finally:
9         print("executing finally clause")
10>>> divide(2, 1)
11 result is 2.0
12 executing finally clause
13>>> divide(2, 0)
14 division by zero!
15 executing finally clause
16>>> divide("2", "1")
17 executing finally clause
18 Traceback (most recent call last):
19   File "<stdin>", line 1, in <module>
20   File "<stdin>", line 3, in divide
21 TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

