Principles of Computer Science II

Errors & Abstract Data Types

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 10

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.

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
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.

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

Exceptions: Examples

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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.

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        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.

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

Last Try statement

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

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    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.

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        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.

```
try:
    raise Exception('spam', 'eggs')
except Exception as inst:
    print(type(inst))    # the exception instance
    print(inst.args)     # arguments stored in .args
    print(inst)          # __str__ allows args to be
        printed directly,
                         # but may be overridden in
                           exception subclasses
    x, y = inst.args     # unpack args
    print('x =', x)
    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.

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

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

User Defined Exceptions: An Example

```
class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error
            occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message
```

User Defined Exceptions: An Example

```
class TransitionError(Error):
    """Raised when an operation attempts a state transition
        that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific
            transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        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.

```
try:
    raise KeyboardInterrupt
finally:
    print('Goodbye, world!')
```

Clean Up Actions: An Example

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and '
    str'
```

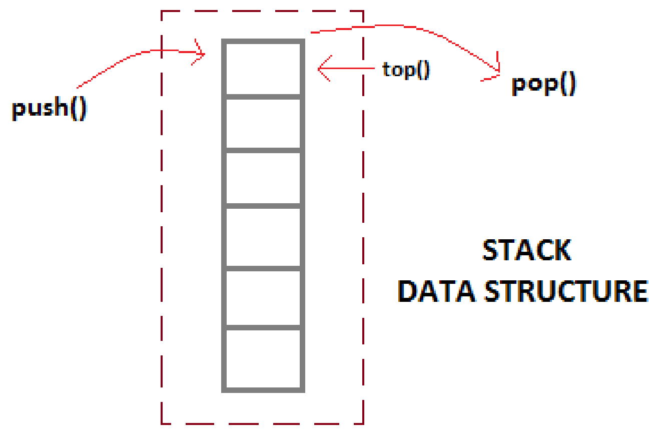
Stacks

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order.

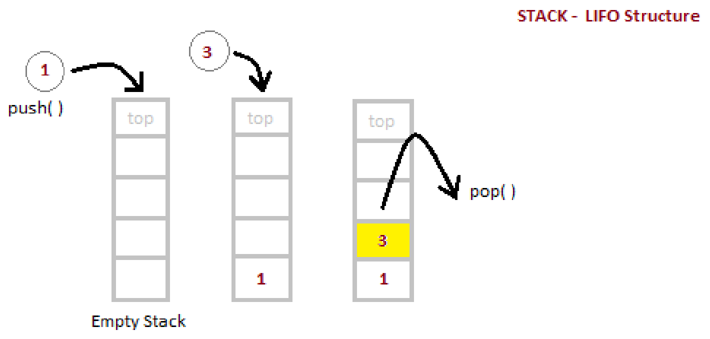
- Every time an element is added, it goes on the top of the stack,
- the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Parsing, Expression Conversion(Infix to Postfix, Postfix to Prefix etc) and many more.

Stacks



Stacks: An Example



In a Stack, all operations take place at the "**top**" of the stack. The "**push**" operation adds an item to the top of the Stack.
The "**pop**" operation removes the item on top of the stack.

Basic features of Stacks

1. Stack is an ordered list of similar data type.
2. Stack is a LIFO structure. (Last in First out).
3. push() function is used to insert new elements into the Stack and
4. pop() function is used to delete an element from the stack.
5. Both insertion and deletion are allowed at only one end of Stack called Top.
6. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

Stacks Code: Initialization

```
class Error(Exception):
    pass

class StackError(Error):
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class Stack(object):
    def __init__(self, size):
        self.content = []
        self.size = size

    def size(self):
        return len(self.content)

    def isEmpty(self):
        return not bool(self.content)
```

Stacks: Algorithm for PUSH operation

1. Check if the stack is full or not.
2. If the stack is full,then print error of overflow and exit the program.
3. If - the stack is not full, then increment the top and add the element.

```
    def push(self, value):
        if len(self.content) >= self.size:
            raise StackError(self, "Overflow")

        self.content.append(value)
```

Stacks: Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

```
    def pop(self):
        if self.content:
            value = self.content.pop()
            return value
        else:
            raise StackError(self, "Empty List")
```

Stacks: Testing

```
if __name__ == '__main__':
    q = Stack(5)

    for i in range(15,20):
        q.push(i)
    for i in range(10,5,-1):
        q.push(i)

    for i in range(1, 13):
        print(q.pop())
```

Stacks: Testing with Error handling

```python
if __name__ == '__main__':
    q = Stack(5)

    try:
        for i in range(15,20):
            q.push(i)
        for i in range(10,5,-1):
            q.push(i)
    except StackError:
        print("Stack is full")

    try:
        for i in range(1, 13):
            print(q.pop())
    except StackError:
        print("Stack is empty")
```