

Principles of Computer Science II

Errors & Abstract Data Types

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 8



Syntax Errors

Until now error messages haven't 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'
```



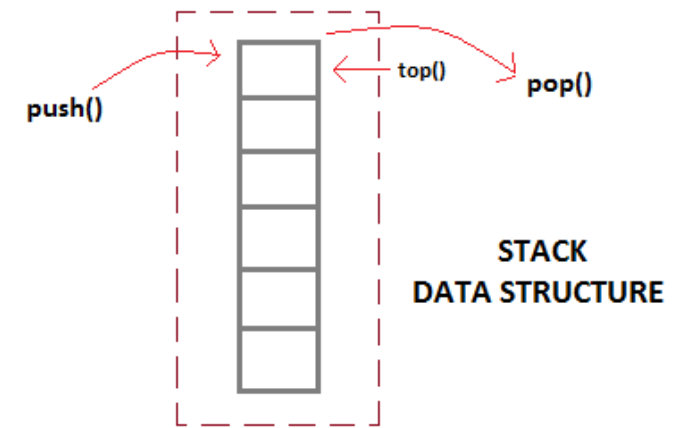
Stacks

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.

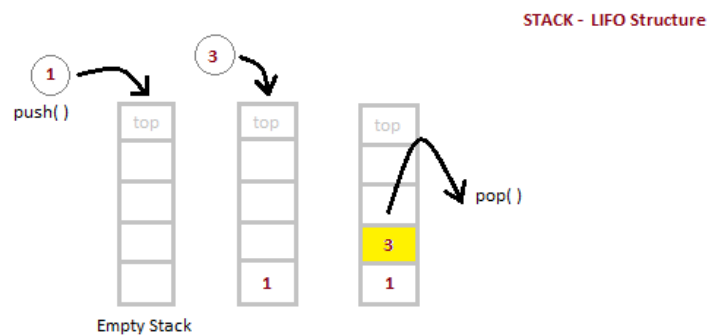
- ▶ Every time an element is added, it goes on the top of the stack,
- ▶ the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.
- ▶ The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - 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

```
1 class Error(Exception):
2     pass
3
4 class StackError(Error):
5     def __init__(self, expression, message):
6         self.expression = expression
7         self.message = message
8
9 class Stack(object):
10    def __init__(self, size):
11        self.content = []
12        self.size = size
13
14    def size(self):
15        return len(self.content)
16
17    def isEmpty(self):
18        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.

```
1     def push(self, value):
2         if len(self.content) >= self.size:
3             raise StackError(self, "Overflow")
4
5         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.

```
1     def pop(self):
2         if self.content:
3             value = self.content.pop()
4             return value
5         else:
6             raise StackError(self, "Empty List")
```



Stacks: Testing

```
1 if __name__ == '__main__':
2     q = Stack(5)
3
4     for i in range(15,20):
5         q.push(i)
6     for i in range(10,5,-1):
7         q.push(i)
8
9     for i in range(1, 13):
10        print(q.pop())
```



Stacks: Testing with Error handling

```
1 if __name__ == '__main__':
2     q = Stack(5)
3
4     try:
5         for i in range(15,20):
6             q.push(i)
7         for i in range(10,5,-1):
8             q.push(i)
9     except StackError:
10        print(" Stack is full")
11
12    try:
13        for i in range(1, 13):
14            print(q.pop())
15    except StackError:
16        print(" Stack is empty")
```



Queues

Queue is also an abstract data type or a linear data structure, in which

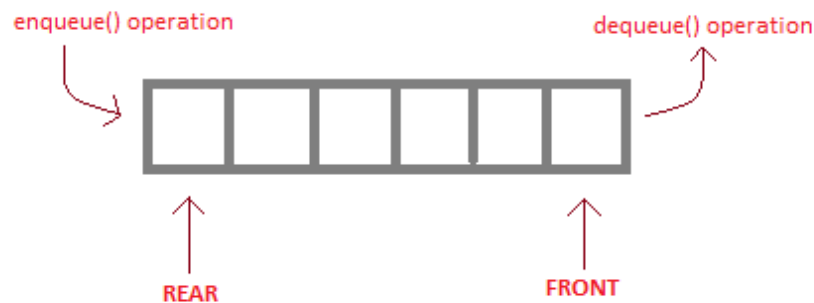
- ▶ the first element is inserted from one end called **REAR** (also called tail),
- ▶ and the deletion of existing element takes place from the other end called as **FRONT**(also called head).

This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will also be removed first.

- ▶ The process to add an element into queue is called Enqueue.
- ▶ the process of removal of an element from queue is called Dequeue.



Queues



enqueue() is the operation for adding an element into Queue.

dequeue() is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE



Basic features of Queues

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO (First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek()** function is oftenly used to return the value of first element without dequeuing it.

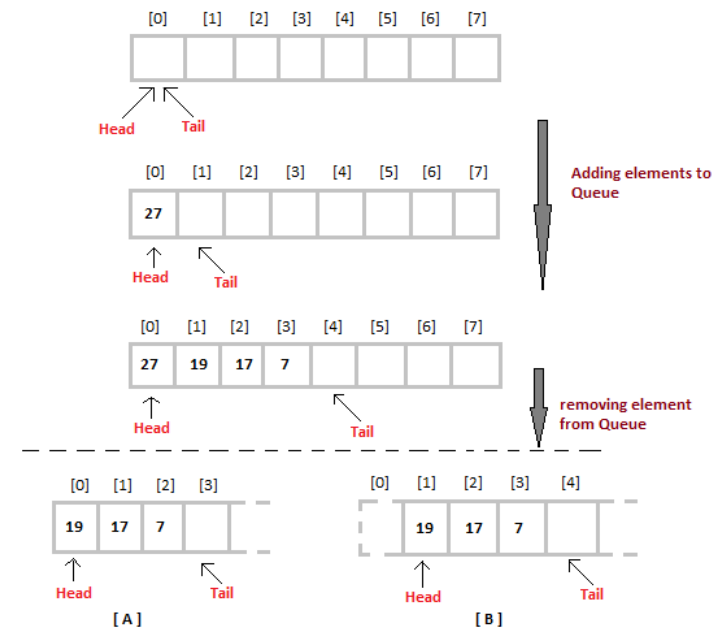


Implementation of Queues

- ▶ Queue can be implemented using an Array, Stack or Linked List.
- ▶ The easiest way of implementing a queue is by using an Array.
- ▶ Initially the **head(FRONT)** and the **tail(REAR)** of the queue points at the first index of the array (starting the index of array from 0).
- ▶ As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.



Queues: An Example



Queues Code: Initialization

```
1 class Error(Exception):
2     """Base class for exceptions in this module."""
3     pass
4
5 class QueueError(Error):
6     def __init__(self, expression, message):
7         self.expression = expression
8         self.message = message
9
10 class Queue(object):
11     def __init__(self):
12         self.content = []
13
14     def size(self):
15         return len(self.content)
16
17     def isEmpty(self):
18         return not bool(self.content)
```



Queues: Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then raise overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

```
1     def enqueue(self, value):
2         return self.content.append(value)
```



Queues: Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then raise underflow error and exit the program.
3. If the queue is not empty, then return the element at the head and increment the head.

```
1 def dequeue(self):
2     if self.content:
3         return self.content.pop()
4     else:
5         raise QueueError("Queue is Empty")
```



Queues: Help function

```
1 def __repr__(self):
2     if self.content:
3         return '{}'.format(self.content)
4     else:
5         return "Queue empty!"
```



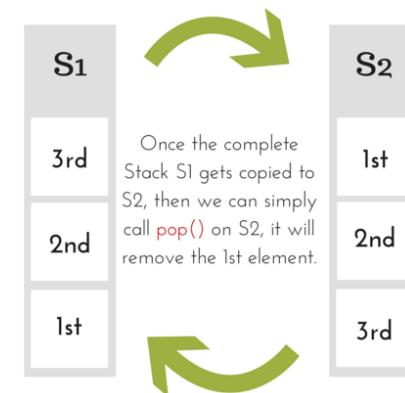
Queues: Testing

```
1 if __name__ == '__main__':
2     queue = Queue()
3     print("Is the queue empty? ", queue.isEmpty())
4     print("Adding 0 to 10 in the queue...")
5     for i in range(10):
6         queue.enqueue(i)
7     print("Queue size: ", queue.size())
8     print("Queue peek: ", queue.peek())
9     print("Dequeue...", queue.dequeue())
10    print("Queue peek: ", queue.peek())
11    print("Is the queue empty? ", queue.isEmpty())
12
13    print("Printing the queue...")
14    print(queue)
```



Queues implemented with Stacks

Pop elements from S1 and push into S2,
`int x = S1.pop();`
`S2.push(x);`



Then push back elements to S1 from S2.



2nd Assignment

<https://www.hackerrank.com/>

- ▶ Complete all **Algorithms** challenges under the following subdomains:
- ▶ Warmup (10), Sorting (15), Recursion (11), Strings (any 9)
- ▶ Total: 45
- ▶ You can cooperate, You can search on the Internet, ...
- ▶ You need to write **your own code**
- ▶ Email `ichatz@dis.uniroma1.it`
Subject: [PCS2] Homework 2
A link to a github repository with your python solutions, for all challenges.
- ▶ **Deadline: 30/November/2017, 23:59**

