Outline

• Generalize Stack/Sack/Queue into Container

• Exceptions
  • custom exceptions
  • try: .... except

• Testing your code
Generalize Stack, Sack, Queue as **Container**

- The methods (add/remove/is_empty) in Container super class
  
  ```python
def __init__(self) -> None:
    raise NotImplementedError("Override this!")
def add(self, obj: object) -> None:
    raise NotImplementedError("Override this!")
def remove(self) -> object:
    raise NotImplementedError("Override this!")
def is_empty(self) -> bool:
    raise NotImplementedError("Override this!")
```

- To give developers the freedom to have different implementations: using **lists**, **dictionaries** in implementing **subclasses** (Stack/Sack/Queue)

- **Container** insures that all **subclasses** will have a **common API between them**, so we can write **client code** that **works with any** stack, sack, or other... Containers

```python
# suppose L is list[Container]
for c in L:
    for i in range(1000):
        c.add(i)
    while not c.is_empty():
        print(c.remove())
```

... so we’ll make Stack, Sack subclasses of Container!
Exceptions - custom exceptions

• They are raised by **ERRORS** detected during execution

• If exceptions are not handled by your program, you may end up getting something like this

• In our implementation of Stack and Sack we want to raise **our own exception** if user try to remove from an empty Container

“EmptyContainerException”
What happens if we did not raise our own exception?

```
>>> from stack import *
>>> s = Stack()
>>> s.remove()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "D:\csc148\lectures\week3\stack.py", line 1,
    return self._contents.pop()
IndexError: pop from empty list
```

If we did not implement our own exception we get IndexError exception
Creating our own exception class

1- Create a class and name it as you like

```python
class EmptyContainerException(Exception):
    """
    Exceptions called when empty Container used inappropriately
    """
    pass
```

2- subclass the Python Exception class

3- we will ignore the implantation for now as we just want to raise the exception and leave handling the error to the Python Exception class
Creating our own exception class

4. import the `EmptyContainerException` class
In our implementation we placed this class in the same file of the Container class

5. raise the exception
In this case we want to raise it if a user tries to remove from empty Stack
Creating our own exception class

```python
>>> from stack import *
>>> s = Stack()
>>> s.remove()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "D:\csc148\lectures\week3\stack.py", line 4
    raise EmptyContainerException
container.EmptyContainerException: Uh-oh, nothing to remove!
```

this is name of the our exception we raised
Exceptions-try: .... except

• Try....except are used to handle exceptions and prevent your code from getting terminated by Python.

• The syntax is as follows:

   try:
       code that may cause error
   except Exception as e:
       print(“error message to inform the user")
       print(e)# show the error
def make_errors() -> None:
    try:
        1/0
        #print(int("hello"))
    except ZeroDivisionError as zd:
        print("Do not divide by Zero Please")
        print(zd)
    except ValueError as ve:
        print("Do not covert a string to int")
        print(ve)
    except Exception as e:
        print("General Exception")
        print(e)
    print("more code in this method")

make_errors()
print("my code continue running normally")
print(42)
Exceptions- try: .... Except--example: 2

# Experiment with exceptions changing what is commented out in the try block

class SpecialException(Exception):
    """class docstring here --- child of Exception"
    pass

class ExtremeException(SpecialException):
    """grandchild of Exception"
    pass
Exceptions- try: .... Except--example: 2

```python
if __name__ == '__main__':
    num = 1, denum=0
    try:
        if (denum==0):
            # raise SpecialException('I am a SpecialException')
            # raise Exception('I am an Exception')
            raise ExtremeException('I am an ExtremeException')
        else:
            print(num/denum)
    # block to run if SpecialException was raised
    # use the name se if one is detected
    except SpecialException as se:
        print(se)
        print('caught as SpecialException')
    except ExtremeException as ee:
        print(ee)
        print('caught as ExtremeException')
    except Exception as e:
        print(e)
        print('caught as Exception')
    print('I am outside try')
    print('my code did not stop due to exception')
```
Testing your code

• You have been using docstring for testing as you develop

• Today we will use: **Python unittest** to make sure that a particular implementation remains consistent with your **ADT's** interface

• What is Python unittest?
  • A framework provided by Python that supports test automation

• How to use unittest?
  • See the slides next
How to use unittest? E.g. StackEmptyTestCase

1- import module unittest and the class you want test (in this example Stack)

2- subclass unittest.TestCase

3- override special methods setUp() tearDown()

4- begin each method that carries out a test with the string test then the method name. e.g testIsEmpty() test the method is_empty() in the stack

5- user assert to check expected outcome assertEqual/assertTrue/assertFalse
Another test case for stack

```python
class StackAllTestCase(unittest.TestCase):
    """Comprehensive tests of (non-empty) Stack."""

    def setUp(self):
        """Set up an empty stack."""
        self.stack = Stack()

    def tearDown(self):
        """Clean up."""
        self.stack = None

    def testAll(self):
        """Test adding and removing multiple elements."""

        for item in range(20):
            self.stack.add(item)
            self.assertTrue(not self.stack.is_empty(),
                            "is_empty() returned True on non-empty Stack!")

        expect = 19
        while not self.stack.is_empty():
            assert self.stack.remove() == expect, \
            ("Something wrong on top of the Stack! Expected ' + \
             str(expect) + '.")
        expect -= 1
```

Tests a range of different values
Output of running unittests

C:\Users\AAA\AppData\Local\Programs\Python\Python36\python.exe
  D:/csc148/lectures/week3/w3/test_stack_unittest.py

Ran 3 tests in 0.001s

OK

Process finished with exit code 0
General remarks when using unittest?

• compose tests before and during implementation

• choosing test cases
  • since you can't test every input, try to think of representative cases:
    • smallest argument(s): 0, empty list or string, ...
    • boundary case: moving from 0 to 1, empty to non-empty, ...
    • "typical" case

• isolate units
  • test classes separately
  • test (related) methods separately
Where Can I find the code presented in class

• You can find the full code in the course website under section **MWF2 (L0301) and MWF3 (L0401)**

• with the following file names:
  • try_except_example.py
  • exceptions.py
  • test_stack_unittest.py
  • test_sack_unittest.py

• Download them Try different things with them and practice
  • Do not be afraid of doing mistakes
Announcements

• A1 due in less than 1 week
• Lab1 marks are posted but the paper_based is not ready yet
• Demo 1: example from last year will be posted soon.