

Programming for Engineers

Lecture 6 - Abstract Data Types

Radoslav Škoviera

Table of contents

Example of a “calculator” using stack and queue	1
Abstract classes	1
Stack	4
Queue	7
Calculator	9
Testing	12
Test stack	12
Test queue	13
Test calculator	14

Example of a “calculator” using stack and queue

Abstract classes

```
from abc import ABC, abstractmethod
from typing import Any
import numpy as np

class Stack(ABC):

    def __init__(self, max_size: int = 128):
        """Create the stack instance that can store a maximum of `max_size` items.
        Args:
            max_size (int, optional): Max size of the stack. Defaults to 128.
        """
        self._max_size = max_size
```

```

def push(self, value: Any):
    """Push an item to the stack. Raises an error if full.
    Args:
        value (Any): The item to push into the stack.
    """
    if len(self) < self.max_size: # if the stack is not full
        self._push_internal(value) # push the item in
    else:
        raise OverflowError("Cannot push, stack full!")

@abstractmethod
def _push_internal(self, value):
    pass # actual push implementation will go here

def pop(self) -> Any:
    """Removes (pops) the item from the top of the stack
    and returns it
    Raises an error if full.

    Returns:
        Any: The item from the top of the stack.
    """
    if not self.empty: # if the stack is not empty
        return self._pop_internal() # get and remove the top-most item
    else:
        raise ValueError("Cannot pop, stack empty!")

@abstractmethod
def _pop_internal(self):
    pass # actual pop implementation will go here

def peek(self) -> Any:
    """Returns the item from the top of the stack without removing it.
    Raises an error if empty.

    Returns:
        Any: The item from the top of the stack.
    """
    if not self.empty: # if the stack is not empty
        return self._peek_internal() # get the top-most item
    else:
        raise ValueError("Cannot peek, stack empty!")

```

```

@abstractmethod
def _peek_internal(self):
    pass # actual peek implementation will go here

@property
def empty(self):
    """Whether there are any items in the stack.
    """
    return len(self) == 0

@property
def full(self):
    """Whether the stack has `max_size` items.
    If `True`, no more items can be pushed into the stack.
    """
    return len(self) == self._max_size

@property
def max_size(self):
    """Max number of items that can be stored in the stack.
    """
    return self._max_size

@abstractmethod
def __len__(self):
    """Number of items in the stack.
    """
    return 0

def __repr__(self):
    # this is just for visualization purposes
    return "AbstractStackClass"

class Queue(ABC):
    """A class implementing the standard ATD Queue
    """

    def __init__(self):
        """Create an empty queue instance
        """
        pass

```

```

@abstractmethod
def push(self, value: Any):
    """Push the `value` into the queue
    """
    pass

@abstractmethod
def pop(self) -> Any:
    """Removes the item from the beginning of the queue
    and returns it
    """
    return 0

@abstractmethod
def peek(self) -> Any:
    """Returns the item from the top of the queue without removing it.
    """
    return 0

@property
@abstractmethod
def empty(self) -> bool:
    """Return True if there are no items in the queue;
    Returns False otherwise.
    """
    return True # TODO: check if the queue is empty

@abstractmethod
def __len__(self):
    """Return the number of items in the queue
    """
    pass

```

Stack

```

class Link():

    def __init__(self, item: Any, previous_link: 'Link | None') -> None:
        self._item = item
        self._previous_link = previous_link

```

```

@property
def item(self):
    return self._item

@property
def previous_link(self):
    return self._previous_link

def free(self):
    self._item = None
    self._previous_link = None

class StackList(Stack):

    def __init__(self, max_size: int = 128):
        super().__init__(max_size)
        self._items = [] # create an empty list

    def _push_internal(self, value):
        self._items.append(value) # simply append the value

    def _pop_internal(self):
        item = self._items[-1] # get the last item
        del self._items[-1] # remove the last item from the list
        return item

    def _peek_internal(self):
        return self._items[-1] # get the last item

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        items = ', '.join([str(i) for i in self._items])
        return f"Stack: {items}"

class StackLink(Stack):

    def __init__(self, max_size: int = 128):
        super().__init__(max_size)

```

```

        self._top = None # _top is the pointer to the latest link (None at first)

def _push_internal(self, value):
    # create a new link and set it to the pushed value
    # link it to the previous `_top` value and "point" `_top` to it
    self._top = Link(value, self._top)

def _pop_internal(self):
    link = self._top # get the last added value from the `_top` pointer
    if link is None: # shouldn't happen, since the "outer" push should catch it
        raise RuntimeError("This should not happen but somehow, the top of stack is None")
    item = link.item # get the item from the top link
    self._top = link.previous_link # set `_top` to the previous link
    link.free() # remove pointers since we are nice programmers
    return item

def _peek_internal(self):
    link = self._top # get the last added value from the `_top` pointer
    if link is None: # shouldn't happen, since the "outer" push should catch it
        raise RuntimeError("This should not happen but somehow, the top of stack is None")
    return link.item # get the item from the top link

def _count_links(self, link: Link):
    if link.previous_link is None: # if this is the last (first added) link
        return 1 # just return 1
    return self._count_links(link.previous_link) + 1 # add 1 to previous result

def __len__(self):
    if self._top is None: # if empty
        return 0 # return zero
    return self._count_links(self._top) # count the links backwards

def _join_values(self, link: Link):
    # similar principle to `_count_links`
    if link.previous_link is None:
        return str(link.item)
    return self._join_values(link.previous_link) + ',' + str(link.item)

def __repr__(self):
    if self._top is None:
        return ''
    return self._join_values(self._top)

```

Queue

```
class QLink():

    def __init__(self, item: Any) -> None:
        self._item = item
        self.next: 'QLink | None' = None

    @property
    def item(self) -> Any:
        return self._item

    def free(self):
        self._item = None
        self.next = None


class QueueList(Queue):

    def __init__(self):
        """Create the queue instance
        """
        self._items = []

    def push(self, value: Any):
        """Push an item to the queue.
        """
        self._items.append(value)

    def pop(self) -> Any:
        """Removes (pops) the item from the top of the queue
        and returns it
        """
        return self._items.pop(0)

    def peek(self) -> Any:
        """Returns the item from the top of the queue without removing it.
        """
        return self._items[0]

    def __len__(self):
        return len(self._items)
```

```

@property
def empty(self):
    """Whether there are any items in the queue.
    """
    return len(self._items) == 0

class QueueLink(Queue):

    def __init__(self):
        self._front = None
        self._back = None

    def push(self, value: Any):
        link = QLink(value)
        if self._front is None:
            self._front = link
            self._back = link
        else:
            if self._back is None:
                raise RuntimeError("Back end of the queue is somehow empty (list link?)")
            self._back.next = link
            self._back = link

    def pop(self) -> Any:
        if self._front is None:
            raise ValueError("Cannot pop, queue is empty!")
        item = self._front.item
        self._front = self._front.next
        return item

    def peek(self) -> Any:
        if self._front is None:
            raise ValueError("Cannot peek, queue is empty!")
        return self._front.item

    def __len__(self):
        if self._front is None:
            return 0

        count = 0
        link = self._front

```



```

    while link is not None:
        count += 1
        link = link.next
    return count

@property
def empty(self):
    return self._front is None

```

Calculator

```

class CalcParser:

    BINARY_FUNCTIONS = { # definitions of binary functions (two operands)
        "+": lambda a, b: a + b,
        "-": lambda a, b: a - b,
        "*": lambda a, b: a * b,
        "/": lambda a, b: a / b,
    }

    UNARY_FUNCTIONS = { # definitions of unary functions (one operand)
        "sin": np.sin,
        "cos": np.cos,
    }

    VARIABLES = { # definitions of predefined variables
        "pi": np.pi,
        "e": np.e,
    }

    PRECEDENCE = { # precedence of operators
        '+': 1,
        '-': 1,
        '*': 2,
        '/': 2,
    }

    STACK_CLASS = StackList
    QUEUE_CLASS = QueueList

    def __init__(self, instruction: str):
        self.postfix_queue: Queue = self.QUEUE_CLASS()
        self.operator_stack: Stack = self.STACK_CLASS()
        tokens = self._tokenize(instruction)

```

```

self._process_tokens(tokens)

def _tokenize(self, instruction: str) -> Queue:
    """ Goes through instruction string and splits it into "tokens"
    - valid operators, operands, and functions.
    """
    tokens: Queue = self.QUEUE_CLASS()
    current_token = ''
    i = 0

    while i < len(instruction): # go through the string
        c = instruction[i] # character-by-character

        if c.isspace(): # if the current character is a space
            if current_token:
                tokens.push(current_token)
                current_token = ''
            i += 1
        elif c in '()+-*/.': # if the current character is a basic operator
            if current_token:
                tokens.push(current_token)
                current_token = ''
            tokens.push(c)
            i += 1
        elif c.isdigit() or c == '.': # if the current character is a number
            current_token += c
            i += 1
        elif c.isalpha(): # if the current character is a letter - function
            current_token += c
            i += 1
            while i < len(instruction) and instruction[i].isalnum():
                current_token += instruction[i]
                i += 1
            tokens.push(current_token)
            current_token = ''
        else:
            print(" " + instruction)
            print(f"{'_'^_':>{i + 3}s}")
            raise ValueError(f"Invalid character: {c} at position {i}")

    if current_token:
        tokens.push(current_token)

```

```

return tokens

def _process_tokens(self, tokens):
    """ Processes a queue of tokens, converting them into a postfix
    expression and storing it in the postfix_queue. This function handles variables,
    unary functions, binary operators, and parentheses to ensure correct
    order of operations.

    Args:
        tokens (Queue): A queue containing tokens from an infix expression.
                        Tokens can be variables, operators, or numeric values.
    """

    while not tokens.empty:
        token = tokens.pop()
        if token in self.VARIABLES:
            self.postfix_queue.push(self.VARIABLES[token])
        elif token in self.UNARY_FUNCTIONS:
            self.operator_stack.push(token)
        elif token == '(':
            self.operator_stack.push(token)
        elif token == ')': # extract the contents of the brackets
            while not self.operator_stack.empty and self.operator_stack.peek() != '(':
                self.postfix_queue.push(self.operator_stack.pop())
            self.operator_stack.pop() # Remove '('
            if not self.operator_stack.empty and self.operator_stack.peek() in self.UNARY_FUNCTIONS:
                self.postfix_queue.push(self.operator_stack.pop())
        elif token in self.PRECEDENCE: # resolve operators with precedence
            while (not self.operator_stack.empty and
                   self.operator_stack.peek() != '(' and
                   self.PRECEDENCE[self.operator_stack.peek()] >= self.PRECEDENCE[token]):
                self.postfix_queue.push(self.operator_stack.pop())
            self.operator_stack.push(token)
        else:
            try:
                self.postfix_queue.push(float(token))
            except ValueError:
                raise ValueError(f"Invalid token: {token}")

    while not self.operator_stack.empty:
        op = self.operator_stack.pop()
        if op == '(':

```

```

        raise ValueError("Mismatched parentheses")
    self.postfix_queue.push(op)

def compute(self):
    """
    Computes the result of a given postfix expression.

    After the expression has been converted into postfix notation and stored in the
    postfix_queue, this function evaluates the expression by going through the queue
    and performing the required operations. The result of the expression is stored in a
    stack and is returned at the end.
    """
    stack = self.STACK_CLASS()
    while not self.postfix_queue.empty():
        token = self.postfix_queue.pop()
        if isinstance(token, (int, float)):
            stack.push(token)
        elif token in self.UNARY_FUNCTIONS:
            a = stack.pop()
            stack.push(self.UNARY_FUNCTIONS[token](a))
        elif token in self.BINARY_FUNCTIONS:
            b = stack.pop()
            a = stack.pop()
            stack.push(self.BINARY_FUNCTIONS[token](a, b))
        else:
            raise ValueError(f"Unknown token: {token}")
    return stack.pop()

```

Testing

Test stack

```

# s = StackList()
s = StackLink()
print(s.empty)
s.push(1)
s.push(2)
print(s)
print(len(s))
print(s.empty)

```

```

print(s.full)
while not s.full:
    s.push(np.random.randint(0, 255))
print(len(s))
print(s.empty)
print(s.full)
while not s.empty:
    ret = s.pop()
    print(ret, end=",")
print()
print(s.empty)

```

```

True
1,2
2
False
False
128
False
True
228,180,198,101,28,139,157,232,22,71,241,13,177,27,153,135,221,74,181,112,173,196,248,250,18
True

```

Test queue

```

queue = QueueLink()
# queue = QueueList()
print(queue.empty)
queue.push(1)
queue.push(2)
queue.push(3)
print(queue.peek())
print(queue.empty)
print(queue.pop())
print(queue.pop())
print(queue.pop())

```

```

True
1

```

False

1

2

3

Test calculator

```
instructions_test = { # test instructions
    "42": 42,
    "1 + 1": 2,
    "2 + 3 * 5": 17,
    "pi": np.pi,
    "e": np.e,
    "2 + 3 * (4 - 5)": -1,
    "sin(pi)": 0,
    "sin(pi * 2)": 0,
    "cos(pi) - 2 + e": np.e + np.cos(np.pi) - 2,
    "sin(pi/2) + cos(0)": 2,
}
for instruction, correct_result in instructions_test.items():
    print(f"Testing:\n{instruction}")
    calc = CalcParser(instruction)
    result = calc.compute()
    assert np.allclose(
        [result],
        [correct_result]
    ), f"The result was {result} but should've been {correct_result}."
    print(f"Correct result! {result: .4f}")
```

Testing:

42

Correct result! 42.0000

Testing:

1 + 1

Correct result! 2.0000

Testing:

2 + 3 * 5

Correct result! 17.0000

Testing:

pi

```
Correct result!  3.1416
Testing:
e
Correct result!  2.7183
Testing:
2 + 3 * (4 - 5)
Correct result! -1.0000
Testing:
sin(pi)
Correct result!  0.0000
Testing:
sin(pi * 2)
Correct result! -0.0000
Testing:
cos(pi) - 2 + e
Correct result! -0.2817
Testing:
sin(pi/2) + cos(0)
Correct result!  2.0000
```