Abstract Data Types

Content:

- What is an abstract data type (ADT)?
- ADT: Stack, Queue, with applications

Abstract Data Types - A User Perspective

An Abstract Data Type (ADT) is a

> ... class of objects whose logical behavior is defined by a set of values and a set of operations.
>
> https://en.wikipedia.org/wiki/Abstract_data_type (https://en.wikipedia.org/wiki/Abstract_data_type)

> ... is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented.
>
> http://interactivepython.org/runestone/static/pythonds/Introduction/WhyStudyDataStructuresandAbs (http://interactivepython.org/runestone/static/pythonds/Introduction/WhyStudyDataStructuresandAb

- *What does this mean?*
- Have we seen and used ADTs?

Lists

We have in fact encountered abstract data types before, see the following example:

In [ ]:

```
1 ringding_lst = []
2 ringding_lst.append('Ring')
3 ringding_lst.append('ding')
4 ringding_lst.append(3)
5 ringding_lst.append('dingeringeding')
6 print(ringding_lst)
```

In [ ]:

```
ringding_lst.remove('ding')
print(ringding_lst)
print(len(ringding_lst))
```

Values?

- A list is a collection of Python *values* in a specific order.

Set of operations?

Lists provide behaviour (methods) such as:

- `l.append(value)` : Add `value` to the end of the collection
- `l.remove(value)` : Remove `value` from the collection

Additionally, the length of a list can be determined with the `len` function

- `len(l)` : The number of elements in the collection

The empty list is represented by `[]` .

User-centric Perspective

- While we did a lot of programming, we actually do not know how a `list` is implemented!
- An ADT describes a *user perspective*, not an implementation perspective.
- It is specified by providing a collection of values that "are the specific ADT", and a list of operations to modify it.

Exercise!

Describe to each other the *dictionary* ADT, that Python provides internally for mapping keys to values.

- What are the values allowed in a dictionary?
- List some methods and describe their behavior in the style of the description for Python's `list` .

Basic Abstract Data Types: Stacks and Queues

Last time we talked about *algorithms* we saw algorithms for finding elements in lists and for sorting elements in lists.

Often, algorithms get easier to describe by modelling the data appropriately.

The Stack ADT

A *Stack* is a list in which we can only add to the end and remove from the end. That is called a *LIFO* list, i.e., last in, first out.

Create a Python module called `my_adts.py` and add the following code.

In [2]:

```
class Stack:

    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def peek(self):
        return self.items[-1]
```

In [ ]:

```
my_stck = Stack()
my_stck.push(5)
```

- What does it do?
- How can you now use a Stack?

In [3]:

```
my_stack = Stack()

my_stack.push('H')
my_stack.push('e')
my_stack.push('l')
my_stack.push('l')
my_stack.push('o')
print(my_stack.peek())
```

o

Specification of the Abstract Data Type (ADT) Stack

Values

- A stack is an ordered collection of items where items are added to and removed from the end called the *top*. Stacks are ordered by the *LIFO* principle.

Methods

Name	Description
`__init__`	Called via `Stack()` , creates an empty stack, i.e., an empty collection of items.
`push(item)`	Adds a new item on top of the stack. The parameter is the item to be added, it returns nothing.
`peek()`	Returns the top item from the stack. It does not modify the stack.
`pop()`	Removes the top item from the stack. No parameter is given, it returns an item. The stack is modified.
`is_empty()`	Returns true, if and only if the stack is empty.
`size()`	Returns the number of items that are stored in the stack.

In [ ]:

```
my_stack = Stack()

my_stack.push('H')
my_stack.push('e')
my_stack.push('j')
my_stack.pop()
my_stack.pop()

my_stack.peek() == 'H'
```

Exercises!

We implemented now the first three methods `__init__` , `push(item)` , and `peek()`

Add the methods `pop()` , `is_empty()` , and `size()` to your implementation of the stack ADT.

In [9]:

```python
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        pass

    def push(self, item):
        self.items.append(item)

    def pop(self):
        last_el = self.items[-1]
        self.items = self.items[:-1]
        return last_el

        # return self.items.pop(-1)

    def peek(self):
        return self.items[-1]

    def size(self):
        pass
```

In [10]:

```python
my_stack = Stack()

my_stack.push('H')
my_stack.push('e')
my_stack.push('j')
el = my_stack.pop()
print(el)
my_stack.pop()

my_stack.peek() == 'H'
```

```
j
```

Out[10]:

```
True
```

In [ ]:

```python

```

Why does this matter?

Now, we can easily build tools that check text for syntactical correctness, i.e., for correctly balanced parenthesis and quote signs.

Consider the following text, an excerpt from *The Hounds of Baskervilles*. It contains many quotation marks, such as “ and ” and we want to help Arthur Conan Doyle with a program to always remember to set closing quotation marks.

In [ ]:

```
def get_text_from_file(path_to_file):
    with open(path_to_file) as fp:
        content_lines = fp.readlines()
    return content_lines


lines = get_text_from_file('the_hound_of_the_baskervilles.txt')
text_for_analysis = ''.join(lines[146:172])
print(text_for_analysis)
```

In [ ]:

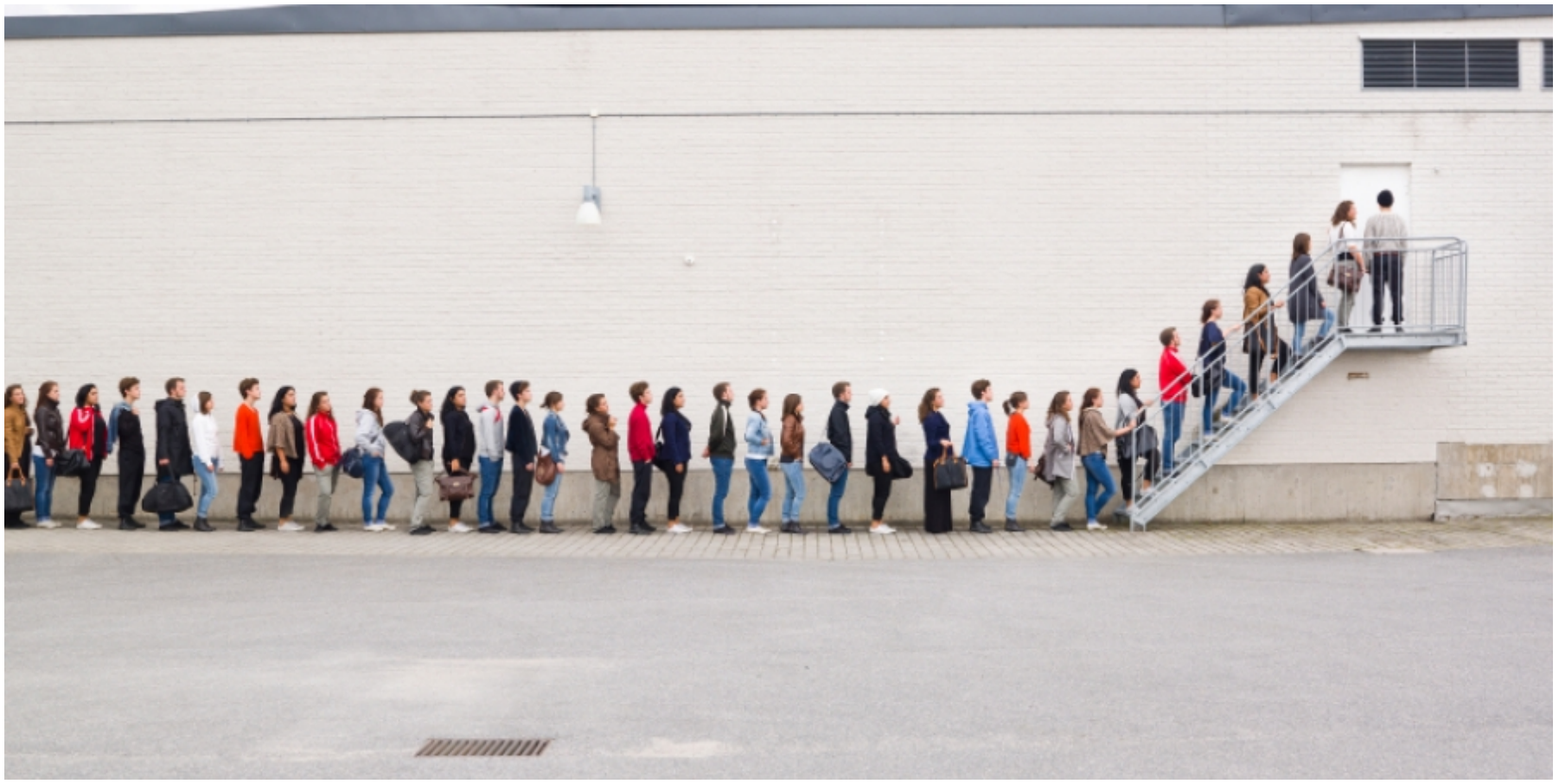
```
def balanced_quotation_marks(text):
    check_stack = Stack()

    for character in text:
        if character == '“':
            check_stack.push(character)
        elif character == '”':
            check_stack.pop()

    if check_stack.is_empty():
        print('I think you quotation signs are balanced.')
        return True
    else:
        print('Hov, it seems as if you forgot to unquote text.')
        return False
```

In [ ]:

```
balanced_quotation_marks(text_for_analysis)
```

What is the runtime of this algorithm?

Specification of the ADT Queue



Values

- A *Queue* is an ordered collection of items which are added at one end, called the *rear*, and removed from the other end, called the *front*. Queues maintain a *FIFO* ordering property.

Methods

Name	**Description**
`__init__()`	creates a new queue that is empty, via a call to `Queue()`.
`enqueue(item)`	Adds a new item to the rear of the queue. It needs an item and returns nothing.
`dequeue()`	Removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
`is_empty()`	Tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
`size()`	Returns the number of items in the queue. It needs no parameters and returns an integer.

Exercise!

Complete the following implementation of the *Queue* ADT.

In [ ]:

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        # TODO: implement me!
        pass

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        # TODO: implement me!
        pass
```

In [ ]:

```

```

What can we do with it now?

We can for example implement a small system helping the ice cream shop next door to keep track of customers waiting and preventing them to jump the line.

In [ ]:

```
from adts import Queue

ismejeriet = Queue()
# Let's say we have 32 customers queueing
for customer_no in range(100, 132):
    ismejeriet.enqueue(customer_no)

# First clerk serving ice cream
print(ismejeriet.dequeue())
# Second clerk serving ice cream
print(ismejeriet.dequeue())
# Third clerk serving ice cream
print(ismejeriet.dequeue())

# In the meanwhile we get two new customers...
ismejeriet.enqueue(132)
ismejeriet.enqueue(133)
# Still they cannot jump the line
print(ismejeriet.dequeue())
```

Abstract Data Types

Content:

- What is an abstract data type (ADT)?
- ADT: Stack, Queue, with applications

Abstract Data Types - A User Perspective

An Abstract Data Type (ADT) is a

> ... class of objects whose logical behavior is defined by a set of values and a set of operations.
>
> https://en.wikipedia.org/wiki/Abstract_data_type (https://en.wikipedia.org/wiki/Abstract_data_type)

> ... is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented.
>
> http://interactivepython.org/runestone/static/pythonds/Introduction/WhyStudyDataStructuresandAbs (http://interactivepython.org/runestone/static/pythonds/Introduction/WhyStudyDataStructuresandAb

- *What does this mean?*
- Have we seen and used ADTs?

Lists

We have in fact encountered abstract data types before, see the following example:

In [ ]:

In [ ]:

Values?

- A list is a collection of Python *values* in a specific order.

Set of operations?

Lists provide behaviour (methods) such as:

- `l.append(value)` : Add `value` to the end of the collection
- `l.remove(value)` : Remove `value` from the collection

Additionally, the length of a list can be determined with the `len` function

- `len(l)` : The number of elements in the collection

The empty list is represented by `[]` .

User-centric Perspective

- While we did a lot of programming, we actually do not know how a `list` is implemented!
- An ADT describes a *user perspective*, not an implementation perspective.
- It is specified by providing a collection of values that "are the specific ADT", and a list of operations to modify it.

Exercise!

Describe to each other the *dictionary* ADT, that Python provides internally for mapping keys to values.

- What are the values allowed in a dictionary?
- List some methods and describe their behavior in the style of the description for Python's `list` .

Basic Abstract Data Types: Stacks and Queues

Last time we talked about *algorithms* we saw algorithms for finding elements in lists and for sorting elements in lists.

Often, algorithms get easier to describe by modelling the data appropriately.

The Stack ADT

A *Stack* is a list in which we can only add to the end and remove from the end. That is called a *LIFO* list, i.e., last in, first out.

Create a Python module called `my_adts.py` and add the following code.

In [2]:

```
```

In [ ]:

```
```

- What does it do?
- How can you now use a Stack?

In [3]:

```
```

o

Specification of the Abstract Data Type (ADT) Stack

Values

- A stack is an ordered collection of items where items are added to and removed from the end called the *top*. Stacks are ordered by the *LIFO* principle.

Methods

Name	Description
`__init__`	Called via `Stack()`, creates an empty stack, i.e., an empty collection of items.
`push(item)`	Adds a new item on top of the stack. The parameter is the item to be added, it returns nothing.
`peek()`	Returns the top item from the stack. It does not modify the stack.
`pop()`	Removes the top item from the stack. No parameter is given, it returns an item. The stack is modified.
`is_empty()`	Returns true, if and only if the stack is empty.
`size()`	Returns the number of items that are stored in the stack.

In [ ]:

Exercises!

We implemented now the first three methods `__init__`, `push(item)`, and `peek()`

Add the methods `pop()`, `is_empty()`, and `size()` to your implementation of the stack ADT.

In [9]:

In [10]:

```
j
```

Out[10]:

```
True
```

In [ ]:

Why does this matter?

Now, we can easily build tools that check text for syntactical correctness, i.e., for correctly balanced parenthesis and quote signs.

Consider the following text, an excerpt from *The Hounds of Baskervilles*. It contains many quotation marks, such as “ and ” and we want to help Arthur Conan Doyle with a program to always remember to set closing quotation marks.

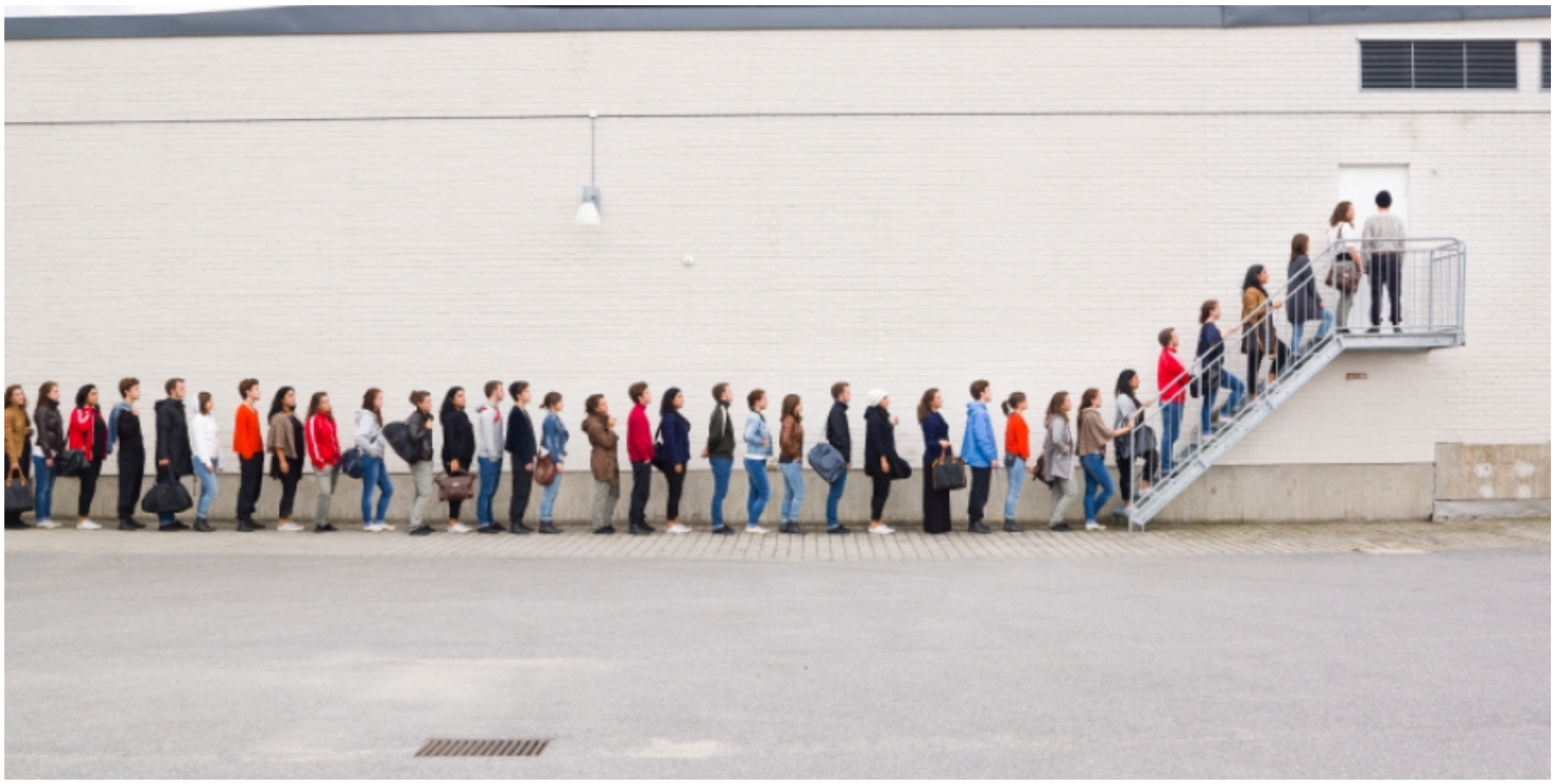
In [ ]:

In [ ]:

In [ ]:

What is the runtime of this algorithm?

Specification of the ADT Queue



Values

- A *Queue* is an ordered collection of items which are added at one end, called the *rear*, and removed from the other end, called the *front*. Queues maintain a *FIFO* ordering property.

Methods

Name	Description
`__init__()`	creates a new queue that is empty, via a call to `Queue()`.
`enqueue(item)`	Adds a new item to the rear of the queue. It needs an item and returns nothing.
`dequeue()`	Removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
`is_empty()`	Tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
`size()`	Returns the number of items in the queue. It needs no parameters and returns an integer.

Exercise!

Complete the following implementation of the *Queue* ADT.

In [ ]:

In [ ]:

What can we do with it now?

We can for example implement a small system helping the ice cream shop next door to keep track of customers waiting and preventing them to jump the line.

In [ ]: