THE UNIVERSITY OF CHICAGO

MASTERS IN COMPUTATIONAL SOCIAL SCIENCE
THE UNIVERSITY OF CHICAGO

MACS 30111

Stacks and queues

Topics:

- ❑ Data types and data structures review
- ❑ Interfaces and APIs
- ❑ Stacks
- ❑ Queues

Data types and data structures review

- Data types: integer, float, string, Boolean
- Data structures: list, dictionary
- Custom data structures

Interface with a Python data structure

We interact with a Python data structure through its ***interface*** without worry about internal implementation details.

The dictionary **interface**:

```
In [1]: d = {}

In [2]: d["A"] = 4.0

In [3]: "A" in d
Out[3]: True
```

Internal implementation:

- Hash table
- Multiple steps:
  - What if a key already exists? What if it doesn’t?
  - What if the hash table doesn’t have enough memory allocated to add more keys?
- Abstract:
  - Interact with the interface
  - Don’t need to think how to manipulate the internal hash table

Interface with a Python module

An API (***Application Programming Interface***) is a collection of functions, protocols, and tools that defines how to interact with a data structure, software library, or system, while abstracting away of the internal details.

E.g., we use the random module API to interact with *random*.

```
In [1]: import random

In [2]: random.randint(1, 100)
Out[2]: 27
```

Twitter API

Topics:

- ❑ Data types and data structures review
- ❑ Interfaces and APIs
- ❑ **Stacks**
- ❑ Queues

Stack data structure

A ***stack*** is a collection of elements with a limited set of operations.

A stack supports the following **operations**:

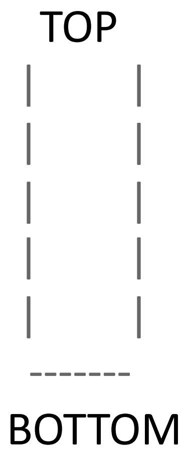
- Create an empty stack
- *Push* a value onto the stack
- *Pop* a value from the stack
- *Peek* at the top of the stack
- Check whether the stack is empty

Example stack

```
  TOP
|  10  |
|  56  |
| 105  |
|  42  |
|   5  |
 -------
 BOTTOM
```

Example

1. Create an empty stack



Example

1. Create an empty stack
2. Push the value 5 to the stack

```
  TOP
 |     |
 |     |
 |     |
 |     |
 |  5  |
 -------
 BOTTOM
```

Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack

```
  TOP
|       |
|  56  |
| 105 |
|  42  |
|   5   |
-------
BOTTOM
```

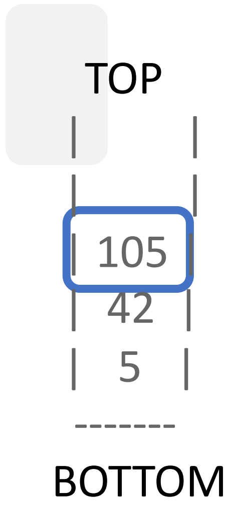
Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack
4. Pop a value from the stack

```
  TOP
|      |
|  56  |
| 105  |
|  42  |
|   5  |
-------
BOTTOM
```

Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack
4. Pop a value from the stack
5. Peek at the stack



TOP

105

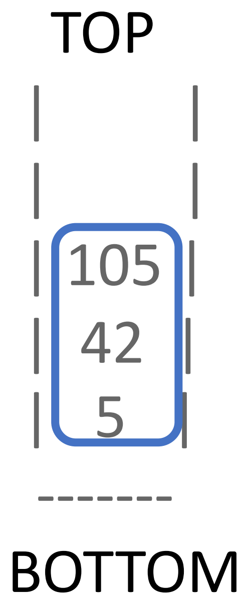
42

5

BOTTOM

Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack
4. Pop a value from the stack
5. Peek at the stack
6. Check whether the stack is empty



TOP

105

42

5

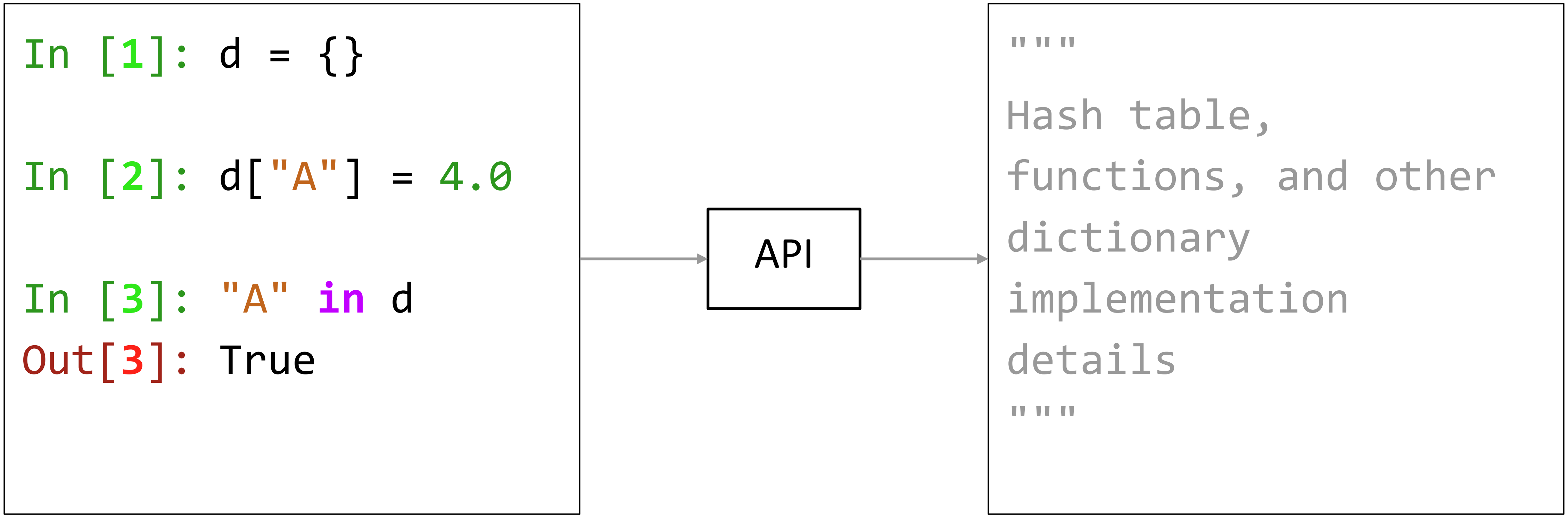
BOTTOM

API revisited

Recall that we interact with a data type in Python through its API.

User interface:

Developer implementation details:



```
In [1]: d = {}

In [2]: d["A"] = 4.0

In [3]: "A" in d
Out[3]: True
```

```
"""
Hash table,
functions, and other
dictionary
implementation
details
"""
```

Stack implementation

Implement a function-based interface for a stack.

```
def stack_create():
    return []
```

Stack implementation

Implement a function-based interface for a stack.

```
def stack_create():
    return []

def stack_push(stack, value):
    stack.append(value)

def stack_pop(stack):
    return stack.pop()

def stack_top(stack):
    return stack[-1]

def stack_is_empty(stack):
    return len(stack) == 0
```

```
  TOP
|     |
| 56  |
| 105 |
| 42  |
|  5  |
-------
BOTTOM
```

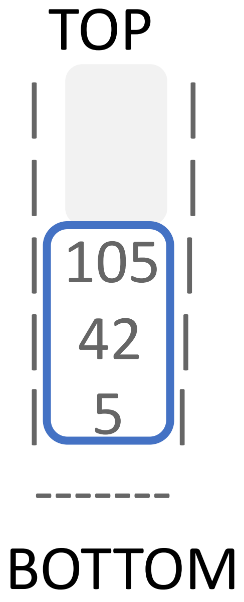
String representation

It’s often a good idea to add a function to visualize a data structure.

```
def stack_to_string(stack):
    s  = " TOP OF THE STACK\n"
    s += "-------------------\n"

    for v in reversed(stack):
        s += str(v).center(20) + "\n"

    s += "-------------------\n"
    s += "BOTTOM OF THE STACK\n"
    return s
```

Example

1. Create an empty stack
2. Push the value 5 to the stack
3. Push the values 42, 105, and 56 to the stack
4. Pop a value from the stack
5. Peek at the stack
6. Check whether the stack is empty



TOP

105

42

5

BOTTOM

First In Last Out

Coding practice: 2.3.2

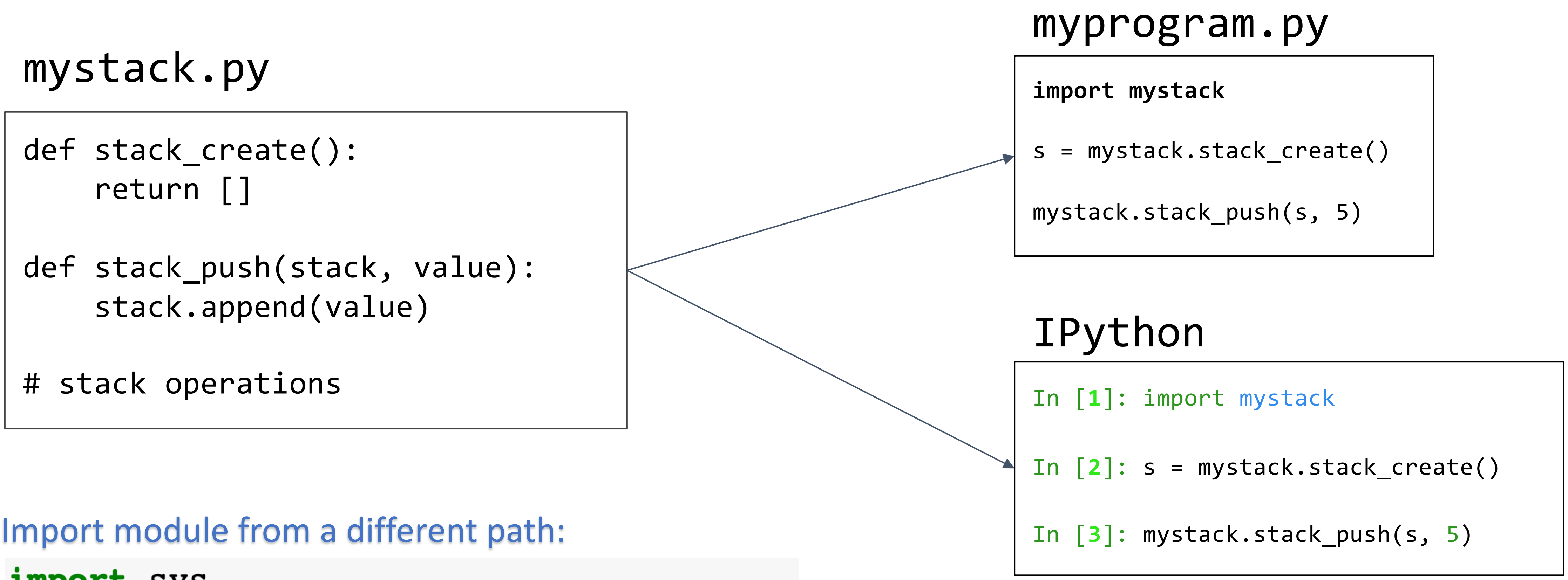
YOUR TURN!!

1. Create an empty stack named `s111`
2. Push the value 42 to the stack
3. Push the values 5, 7, and 12 to the stack
4. Pop a value from the stack
5. Pop a value from the stack
6. Peek at the stack
7. Check whether the stack is empty

```
def stack_create():
    return []

def stack_push(stack, value):
    stack.append(value)

def stack_pop(stack):
    return stack.pop()

def stack_top(stack):
    return stack[-1]

def stack_is_empty(stack):
    return len(stack) == 0
```

Python modules

Once we define an API for a data structure, we can put it in a **Python module** and **import** it from IPython or other Python files.



mystack.py

```
def stack_create():
    return []

def stack_push(stack, value):
    stack.append(value)

# stack operations
```

myprogram.py

```
import mystack

s = mystack.stack_create()

mystack.stack_push(s, 5)
```

IPython

```
In [1]: import mystack

In [2]: s = mystack.stack_create()

In [3]: mystack.stack_push(s, 5)
```

Import module from a different path:

```
import sys
sys.path.append("/path/to/my/modules/")
import my_module
```

Topics:

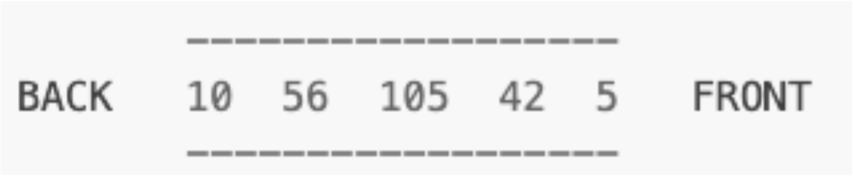
- ❑ Data types and data structures review
- ❑ Interfaces and APIs
- ❑ Stacks
- ❑ **Queues**

Queue data structure

A ***queue*** is a collection of elements with a limited set of operations.

Queue **operations**:

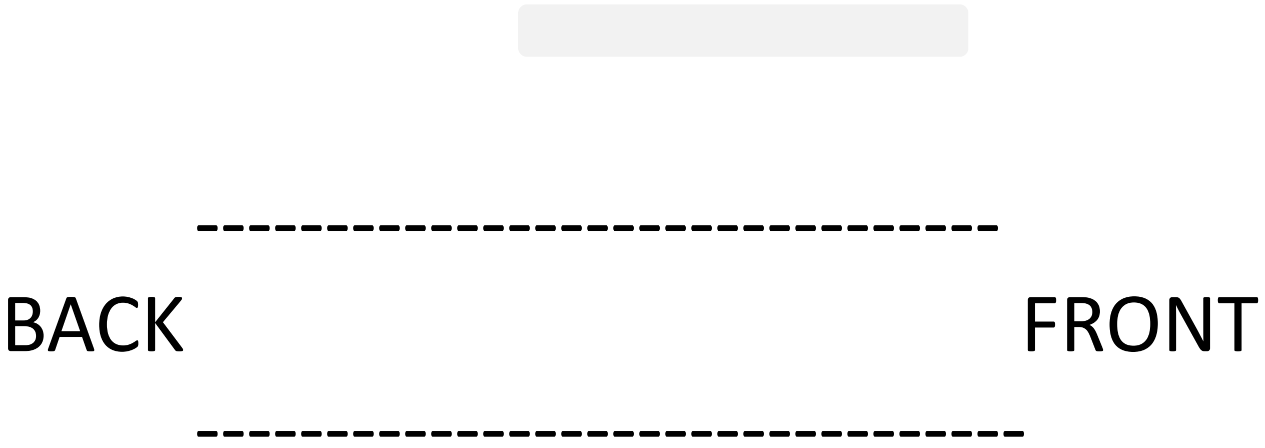
- Create an empty queue
- *Enqueue* a value at the back of the queue
- *Dequeue* a value from the front of the queue
- *Peek* at the front of the queue
- Check the size of the queue



```
           -----------------
BACK     10   56   105   42   5     FRONT
           -----------------
```

Example

1. Create an empty queue



BACK

FRONT

Example

1. Create an empty queue
2. Enqueue the value 5 to the queue

```
      --------------------------------
BACK                             5    FRONT
      --------------------------------
```

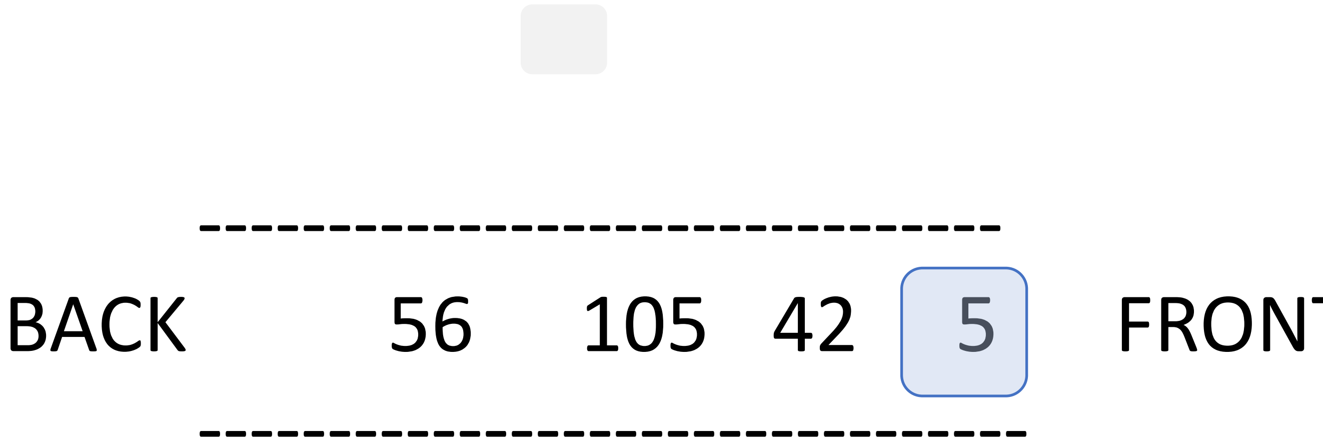
Example

1. Create an empty queue
2. Enqueue the value 5 to the queue
3. Enqueue the values 42, 105, and 56 to the queue

```
       ----------------------------------
BACK        56     105   42     5     FRONT
       ----------------------------------
```

Example

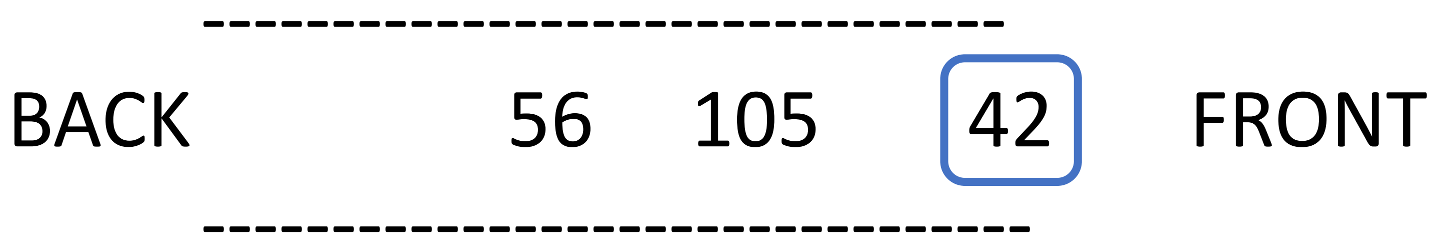
1. Create an empty queue
2. Enqueue the value 5 to the queue
3. Enqueue the values 42, 105, and 56 to the queue
4. Dequeue a value from the queue



BACK 56 105 42 5 FRONT

Example

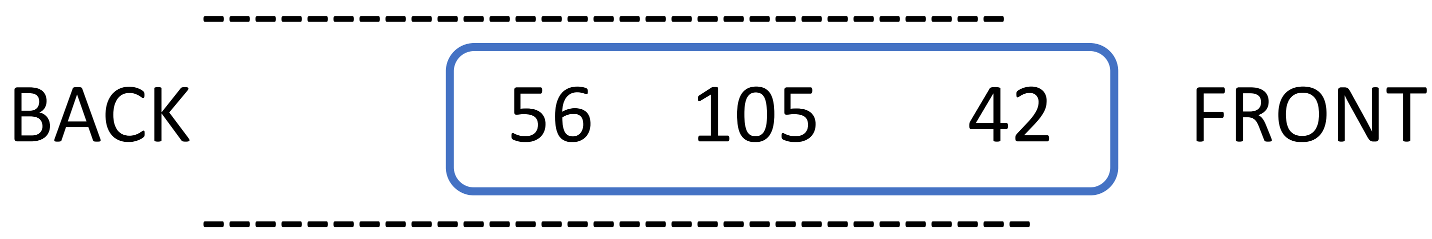
1. Create an empty queue
2. Enqueue the value 5 to the queue
3. Enqueue the values 42, 105, and 56 to the queue
4. Dequeue a value from the queue
5. Peek at the front of the queue



BACK 56 105 42 FRONT

Example

1. Create an empty queue
2. Enqueue the value 5 to the queue
3. Enqueue the values 42, 105, and 56 to the queue
4. Dequeue a value from the queue
5. Peek at the front of the queue
6. Check the size of the queue



BACK 56 105 42 FRONT

Queue implementation

We will implement a queue as a list in Python.

Should the front of the queue be the end of the list?

```
    q = [56, 105, 42, 5]
```

Should it be the beginning of the list?

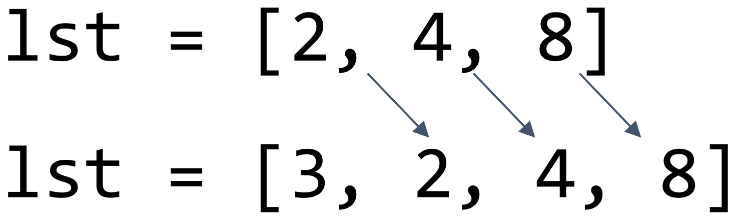
```
    q = [5, 42, 105, 56]
```

```
                 ------------------------------
BACK         56     105   42     5     FRONT
                 ------------------------------
```

Complexity and efficiency of list operations

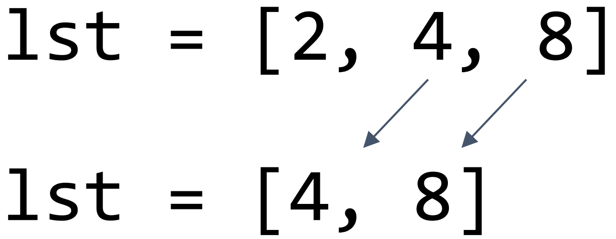
Inserting or deleting **from the beginning** of a list is an **expensive** operation since all elements must be shifted.

Insert the value 3 to the beginning of the list:



```
lst = [2, 4, 8]

lst = [3, 2, 4, 8]
```

Delete the value from the beginning of the list:



```
lst = [2, 4, 8]

lst = [4, 8]
```

https://wiki.python.org/moin/TimeComplexity

Complexity and efficiency of list operations

Appending or deleting **from the end** of a list is a **simple** operation.

Append the value 3 to the end of the list:

```
    lst = [2, 4, 8]

    lst = [2, 4, 8, 3]
```

Delete a value from the end of the list:

```
    lst = [2, 4, 8]

    lst = [2, 4]
```

https://wiki.python.org/moin/TimeComplexity

Queue implementation

What if we make the *beginning* of the list the *back* of the queue?

Enqueue the value 2:

back `[8, 9, 17]` front

`[2, 8, 9, 17]`

Dequeue a value:

`[2, 8, 9, 17]`

`[2, 8, 9]`

What if we make the *beginning* of the list the *front* of the queue?

Enqueue the value 2:

front `[17, 9, 8]` back

`[17, 9, 8, 2]`

Dequeue a value:

`[17, 9, 8, 2]`

`[9, 8, 2]`

Queue implementation

```
              --------------------------------
BACK    56    105   42    5     FRONT
              ---------------------------------
```

Now we can implement a function-based interface for a queue.

```
def queue_create():
return []

def queue_is_empty(queue):
return len(queue) == 0

def queue_length(queue):
return len(queue)

def queue_enqueue(queue, value):
queue.append(value)

def queue_dequeue(queue):
return queue.pop(0)

def queue_front(queue):
return queue[0]
```

```
def queue_to_string(queue):
s = "FRONT OF THE QUEUE\n"
s += "------------------\n"

for v in queue:
s += str(v).center(19) + "\n"

s += "------------------\n"
s += "BACK OF THE QUEUE \n"
return s
```

YOUR TURN!!

1. Create an empty queue named `q111`
2. Enqueue the value 42 to the stack
3. Enqueue the values 5, 7, and 12 to the queue
4. Dequeue a value from the queue
5. Peek at the front of the queue
6. Check the size of the queue
7. Check whether the queue is empty

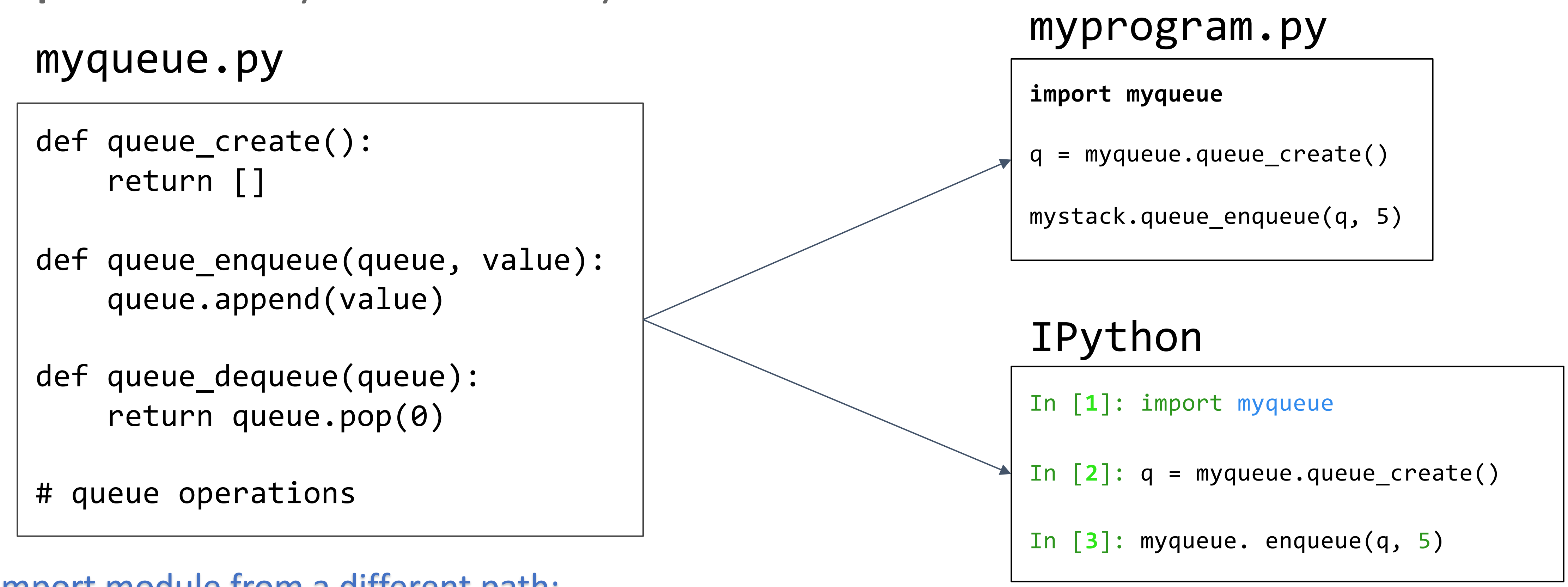
```
def queue_create():
return []

def queue_is_empty(queue):
return len(queue) == 0

def queue_length(queue):
return len(queue)

def queue_enqueue(queue,
value):
queue.append(value)

def queue_dequeue(queue):
return queue.pop(0)

def queue_front(queue):
return queue[0]
```

First In First Out vs Last In First Out

Coding practice: 2.3.3

Python modules

Once we define an API for a data structure, we can put it in a **Python module** and **import** it from IPython or other Python files.



myqueue.py

```
def queue_create():
    return []

def queue_enqueue(queue, value):
    queue.append(value)

def queue_dequeue(queue):
    return queue.pop(0)

# queue operations
```

myprogram.py

```
import myqueue

q = myqueue.queue_create()

mystack.queue_enqueue(q, 5)
```

IPython

```
In [1]: import myqueue

In [2]: q = myqueue.queue_create()

In [3]: myqueue. enqueue(q, 5)
```

Import module from a different path:

```
import sys
sys.path.append("path-to/location/”)
import myqueue
```

Recap:

- Data structures:
  - Stacks: when you need last in, first out. Specific methods to add to/from the TOP
  - Queues: when you need first in, first out. Specific methods to add to the BACK of the queue
- Working with files!