

Big Idea™: Interface *vs* Implementation

Interface

What something does

Example: a stack

- **create** an empty stack
- **push** an item to top
- **pop** an item from top
- get stack **length** (size)

```
s = makeStack()  
push(s, 42)  
print(len(s))      # 1  
value = pop(s)  
print(value)       # 42  
print(len(s))      # 0
```

Interface design

**Python interface
(client code)**

Big Idea™: Interface *vs* Implementation

Interface

What something does

Example: a stack

- **create** an empty stack
- **push** an item to top
- **pop** an item from top
- get stack **length** (size)

```
s = makeStack()  
push(s, 42)  
print(len(s))      # 1  
value = pop(s)  
print(value)       # 42  
print(len(s))      # 0
```

Implementation

How it's done

Example: a stack

- How to store elements?
- What is an empty stack?
- Where is top of stack?

Encoding

**how to use preexisting data structures
to implement new data structure**

Big Idea™: Interface *vs* Implementation

Interface

What something does

Example: a stack

- **create** an empty stack
- **push** an item to top
- **pop** an item from top
- get stack **length** (size)

```
s = makeStack()  
push(s, 42)  
print(len(s))      # 1  
  
value = pop(s)  
  
print(value)       # 42  
print(len(s))      # 0
```

Implementation

How it's done

Example: a stack

- store items in a **list**
- top is **end** of list

Encoding

**how to use preexisting data structures
to implement new data structure**

Big Idea™: Interface *vs* Implementation

Interface

What something does

Example: a stack

- **create** an empty stack
- **push** an item to top
- **pop** an item from top
- get stack **length** (size)

```
s = makeStack()  
push(s, 42)  
  
print(len(s))    # 1  
  
value = pop(s)  
  
print(value)    # 42  
print(len(s))    # 0
```

Implementation

How it's done

Example: a stack

- store items in a **list**
- top is **end** of list

```
def makeStack():  
    return []  
  
def push(stack, item):  
    return stack.append(item)  
  
def pop(stack):  
    return stack.pop()
```

Big Idea™: Interface *vs* Implementation

Interface

What something does

Example: a stack

- **create** an empty stack
- **push** an item to top
- **pop** an item from top
- get stack **length** (size)

```
s = makeStack()  
push(s, 42)  
  
print(len(s))    # 1  
  
value = pop(s)  
  
print(value)    # 42  
print(len(s))    # 0
```

Implementation

How it's done

Example: a stack

- store items in a **list**
- top is **beginning** of list

*We can change implementation
without changing interface!*

```
def makeStack():  
    return []  
  
def push(stack, item):  
    return stack.insert(0, item)  
  
def pop(stack):  
    return stack.pop(0)
```

Drawbacks of procedural implementation?

Client code

```
import stack1
import stack2

s1 = stack1.makeStack()
stack1.push(s1, 1)
stack1.push(s1, 2)

s2 = stack2.makeStack()
stack2.push(s2, 3)
stack2.push(s2, 4)

value = stack2.pop(s1)

print(value)
```

Name

1.11 / 13

Implementations

```
def makeStack():
    return []

def push(stack, item):
    return stack.append(item)

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

[stack1.py](#)

```
def makeStack():
    return []

def push(stack, item):
    return stack.insert(0, item)

def pop(stack):
    return stack.pop(0)
```

[stack2.py](#)

Drawbacks of procedural implementation?

Client code

```
import stack1
import stack2

s1 = stack1.makeStack()
stack1.push(s1, 1)
stack1.push(s1, 2)

s2 = stack2.makeStack()
stack2.push(s2, 3)
stack2.push(s2, 4)

value = stack2.pop(s1)
print(value)
```

implementation details can "leak",
causing data and behavior to become separated



Implementations

```
def makeStack():
    return []

def push(stack, item):
    return stack.append(item)

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

[stack1.py](#)

```
def makeStack():
    return []

def push(stack, item):
    return stack.insert(0, item)

def pop(stack):
    return stack.pop(0)
```

[stack2.py](#)

programs
=
data + behavior

encapsulation
(separate interface
from implementation)

object

- combines data & behavior
- access only through interface
- knows about itself
(can access its own data and behavior)

an object is sort-of
like a little state machine!

Object-oriented programming languages **differ** in:

- how the programmer specifies an object's **interface**
- how the programmer specifies an object's **implementation**
- how objects are **created, initialized, queried, and updated**
- **encapsulation** mechanism
how strictly the language *enforces* the separation between interface & implementation

Procedural *vs* object-oriented clients in Python

Procedural

```
import stack1
import stack2
```

```
s1 = stack1.makeStack()
stack1.push(s1, 1)
stack1.push(s1, 2)
```

```
s2 = stack2.makeStack()
stack2.push(s2, 3)
stack2.push(s2, 4)
```

```
value = stack2.pop(s1)
```

```
print(value)
```

implementation details can "leak",
causing data and behavior to become separated

Object-oriented

```
import stack1
import stack2
```

```
s1 = stack1.Stack()
s1.push(1)
s1.push(2)
```

create an instance
of the Stack class
(i.e., "instantiate")

```
s2 = stack2.Stack()
s2.push(3)
s2.push(4)
```

```
value = s1.pop()
```

```
print(value)
```

object stores its own data,
method operates on the object's data

Classes are patterns for objects

A class is like

a cookie cutter



ecx.images-amazon.com/images/I/21owTyO6HaL.jpg

Objects are like

cookies



eclecticrecipes.com/wp-content/uploads/2013/02/heart-6.jpg



images.edge-generalmills.com/9b6a8635-686e-4b7d-863b-7dd3d8d25a04.jpg

A procedural stack implementation

Encoded as a list, where the top of the stack is at the end of the list.

```
def makeStack():  
    '''Creates a new stack'''  
    return []
```

```
def push(stack, item):  
    '''Push item onto top of stack'''  
    stack.append(item)
```

```
def pop(stack):  
    '''Removes and returns the item on top of stack'''  
    return stack.pop()
```

The next several slides transform the procedural implementation to an OO implementation.

An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```
class Stack:
```

```
    '''A LIFO data structure'''
```

```
def makeStack():
```

```
    '''Creates a new stack'''
```

```
    return []
```

```
def push(stack, item):
```

```
    '''Push item onto top of stack'''
```

```
    stack.append(item)
```


```
def pop(stack):
```

```
    '''Removes and returns the item on top of stack'''
```

```
    return stack.pop()
```

Not yet correct

encoding should
be a data attribute
(instance initialization,
not instance creation)



An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```
class Stack:
```

```
    '''A LIFO data structure'''
```

Not yet correct

```
def makeStack():
```

```
    '''Initializes a new stack'''
```

```
    values = []
```



values is a
local variable :(

```
def push(item):
```

```
    '''Push item onto top of stack'''
```

```
    values.append(item)
```

```
def pop():
```

```
    '''Removes and returns the item on top of stack'''
```

```
    return values.pop()
```

An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```
class Stack:
```

```
    '''A LIFO data structure'''
```

```
def makeStack(self):
```

```
    '''Initializes a new stack'''
```

```
    self.values = []
```

```
def push(self, item):
```

```
    '''Push item onto top of stack'''
```

```
    self.values.append(item)
```

```
def pop(self):
```

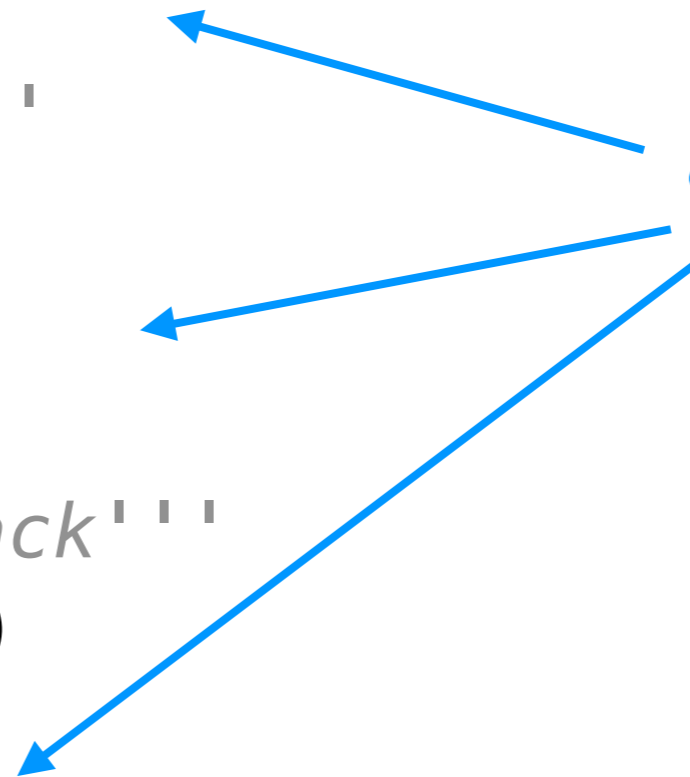
```
    '''Removes and returns the item on top of stack'''
```

```
    return self.values.pop()
```

Not yet correct

explicit self

the name "self"
is a convention



An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```
class Stack:
```

```
    '''A LIFO data structure'''
```

```
def __init__(self):
```

```
    '''Initializes a new stack'''
```

```
    self.values = []
```

```
def push(self, item):
```

```
    '''Push item onto top of stack'''
```

```
    self.values.append(item)
```

```
def pop(self):
```

```
    '''Removes and returns the item on top of stack'''
```

```
    return self.values.pop()
```

Correct,
but not idiomatic

← constructor

`__init__` is a
"special method" that
Python calls automatically,
to initialize an instance.

An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```
class Stack:
```

```
    '''A LIFO data structure'''
```

```
def __init__(self):
```

```
    '''Initializes a new stack'''
```

```
    self._values = []
```

```
def push(self, item):
```

```
    '''Push item onto top of stack'''
```

```
    self._values.append(item)
```

```
def pop(self):
```

```
    '''Removes and returns the item on top of stack'''
```

```
    return self._values.pop()
```

"private" field

By convention, if a member begins with `_`, it's not part of the interface. Clients shouldn't access it directly.

An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```
class Stack:
```

```
    '''A LIFO data structure'''
```

```
def __init__(self):
```

```
    '''Initializes a new stack'''
```

```
    self._values = []
```

```
def push(self, item):
```

```
    '''Push item onto top of stack'''
```

```
    self._values.append(item)
```

```
def pop(self):
```

```
    '''Removes and returns the item on top of stack'''
```

```
    return self._values.pop()
```

Client code

```
s = Stack()  
s.push(1)  
print(s.pop())
```

An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```
class Stack:
```

```
    '''A LIFO data structure'''
```

```
def __init__(self):
```

```
    '''Initializes a new stack'''
```

```
    self._values = []
```

```
def push(self, item):
```

```
    '''Push item onto top of stack'''
```

```
    self._values.append(item)
```

```
def pop(self):
```

```
    '''Removes and returns the item on top of stack'''
```

```
    return self._values.pop()
```

```
def len(self):
```

```
    '''Returns the number of elements in the stack'''
```

```
    return len(self._values)
```

Correct,
but not idiomatic

Client code

```
s = Stack()  
s.push(1)  
print(s.len())
```

An object-oriented stack implementation

Encoded as a list, where the top of the stack is at the front of the list.

```
class Stack:
```

```
    '''A LIFO data structure'''
```

```
def __init__(self):
```

```
    '''Initializes a new stack'''
```

```
    self._values = []
```

```
def push(self, item):
```

```
    '''Push item onto top of stack'''
```

```
    self._values.append(item)
```

```
def pop(self):
```

```
    '''Removes and returns the item on top of stack'''
```

```
    return self._values.pop()
```

```
def __len__(self):
```

```
    '''Returns the number of elements in the stack'''
```

```
    return len(self._values)
```

Client code

```
s = Stack()  
s.push(1)  
print(len(s))
```

python

file / module / session

```
1 class Stack:  
2 ...  
3  
4 s = Stack()
```



scopes

(determined by program code)



2



built-in

print →  (and others)

1

global

Stack →  

s →  
instance of

namespaces

(a snapshot of program execution)

python

file / module / session

```
1 class Stack:  
2 ...  
3  
4 s = Stack()  
5 s.x = 42  
6 print(s.x)  
7 print(s.y)
```



scopes

(determined by program code)



2


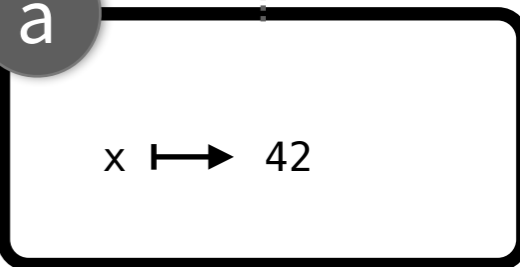
built-in

print \mapsto  (and others)

1

global

Stack \mapsto  

s \mapsto  
x \mapsto 42

instance of

namespaces

(a snapshot of program execution)

python

file / module / session

```
1 class Stack:
2
3     def makeStack():
4         values = [] ← Remember: values is local :(
5
6     def push(item):
7         values.append(item)
8
9     def pop():
10        return values.pop()
11
12 s = Stack()
13 s.makeStack()
14 s.push(42)
```

scopes

(determined by program code)



Not how Python works

2

built-in

print → (and others)

1

global

Stack → **b**

makeStack →

push →

pop →

instance of

s → **a**

0

local

values → []

s.makeStack, called @ line 13

namespaces

(a snapshot of program execution)

python

file / module / session

```
1 class Stack:
2
3     def __init__(self):
4         self._values = []
5
6     def push(self, item):
7         self._values.append(item)
8
9     def pop(self):
10        return self._values.pop()
11
12 s = Stack()
13 s.push(42)
```

scopes

(determined by program code)



How Python works

2

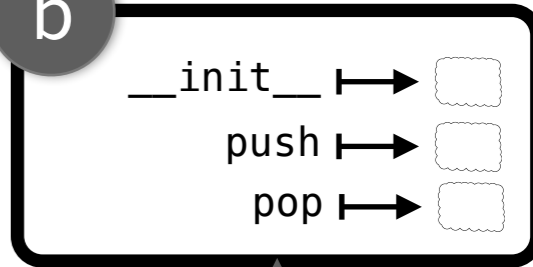
built-in




print →  (and others)

1

global

Stack → **b**



__init__ → 
push → 
pop → 


instance of

a



0

local

self → 

s.__init__, called @ line 12

namespaces

(a snapshot of program execution)

python

file / module / session

```
1 class Stack:
2
3     def __init__(self):
4         self._values = []
5
6     def push(self, item):
7         self._values.append(item)
8
9     def pop(self):
10        return self._values.pop()
11
12 s = Stack()
13 s.push(42)
```

scopes

(determined by program code)



How Python works

2

built-in




print →  (and others)

1

global

Stack → **b**

b

- __init__ → 
- push → 
- pop → 

instance of

a

_values → []

0

local

self →

s.__init__, called @ line 12

namespaces

(a snapshot of program execution)

python

file / module / session

```
1 class Stack:  
2  
3     def __init__(self):  
4         self._values = []  
5  
6     def push(self, item):  
7         self._values.append(item)  
8  
9     def pop(self):  
10        return self._values.pop()  
11  
12 s = Stack()  
13 s.push(42)
```

scopes

(determined by program code)



How
Python works

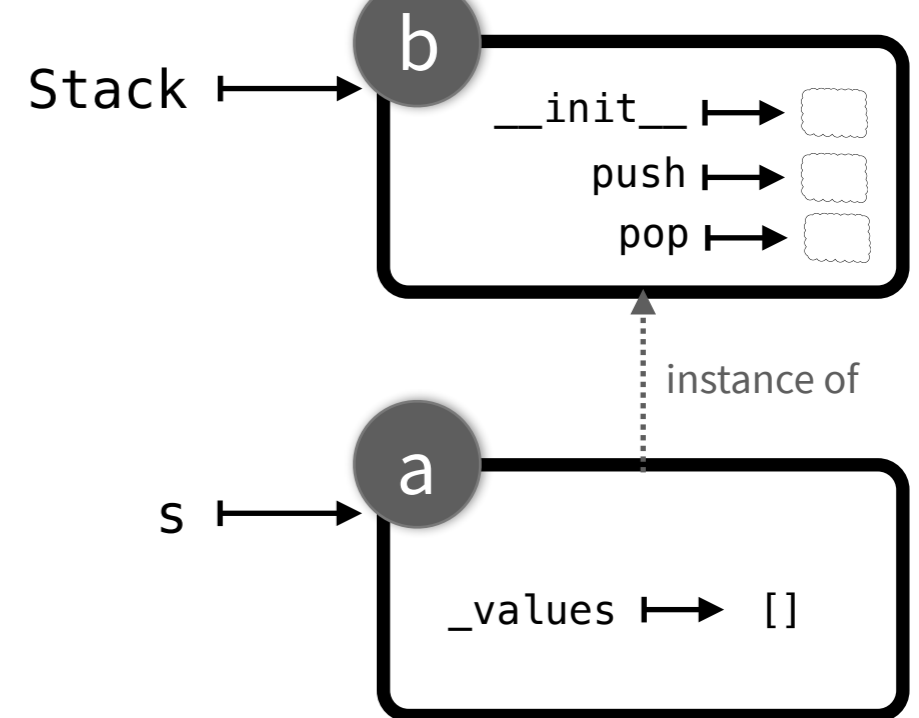
2

built-in

print →  (and others)

1

global



namespaces

(a snapshot of program execution)

python

file / module / session

```
1 class Stack:
2
3     def __init__(self):
4         self._values = []
5
6     def push(self, item):
7         self._values.append(item)
8
9     def pop(self):
10        return self._values.pop()
11
12 s = Stack()
13 s.push(42)
```

scopes

(determined by program code)



How Python works


2

built-in




print →  (and others)


1

global

Stack →  **b**

b

- __init__ → 
- push → 
- pop → 

s →  **a**


a

- _values → []

instance of

0

local

self → 

item → 42

s.__init__, called @ line 12

namespaces

(a snapshot of program execution)

python

file / module / session

```
1 class Stack:
2
3     def __init__(self):
4         self._values = []
5
6     def push(self, item):
7         self._values.append(item)
8
9     def pop(self):
10        return self._values.pop()
11
12 s = Stack()
13 s.push(42)
```

scopes

(determined by program code)



How Python works

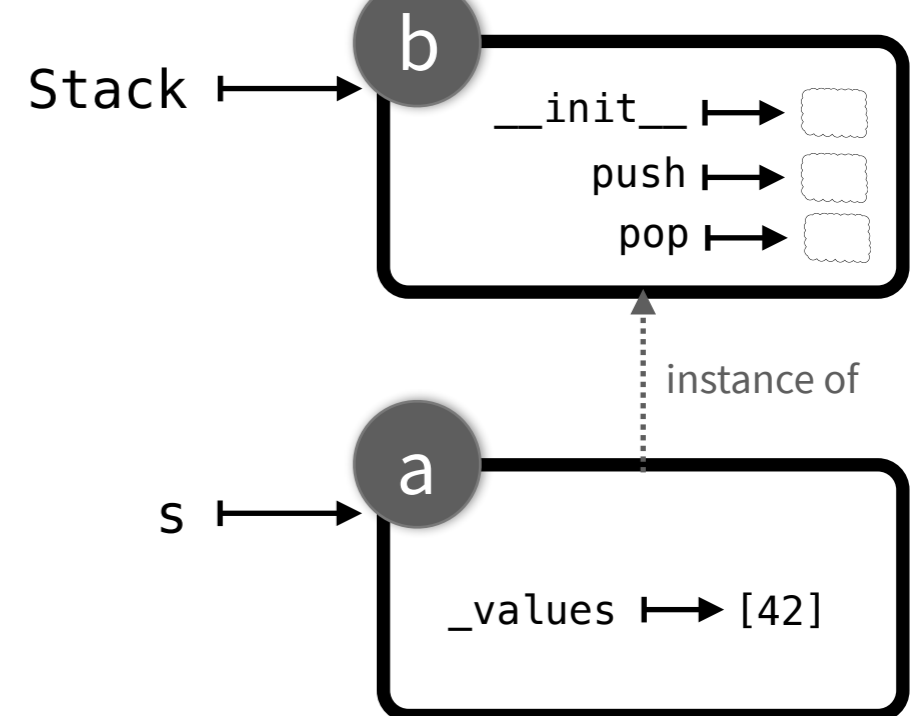
2

built-in

print →  (and others)

1

global



namespaces

(a snapshot of program execution)

Python tutor: pythontutor.com/visualize.html

The screenshot shows the Python Tutor interface. On the left, a code editor displays Python 3.6 code for a Stack class. The code is as follows:

```
1 class Stack:
2
3     def __init__(self):
4         self._values = []
5
6     def push(self, item):
7         self._values.append(item)
8
9     def pop(self):
10        return self._values.pop()
11
12 s = Stack()
13 s.push(42)
```

Line 7 is highlighted with a green arrow, indicating it has just executed. Line 8 is highlighted with a red arrow, indicating it is the next line to execute. Below the code is a legend: a green arrow for 'line that has just executed' and a red arrow for 'next line to execute'. There is also a note: 'Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.' Below this is a progress bar and navigation buttons: '<< First', '< Back', 'Step 9 of 9', 'Forward >', and 'Last >>'. At the bottom, it says 'Created by @pgbovine. Support with a small donation.'

On the right, a memory diagram titled 'Frames' and 'Objects' illustrates the state of memory. The 'Global frame' contains a variable 'Stack' pointing to the 'Stack class' and a variable 's' pointing to a 'Stack instance'. The 'Stack class' has three methods: '__init__' (function __init__(self)), 'pop' (function pop(self)), and 'push' (function push(self, item)). The 'Stack instance' has one attribute: '_values', which points to a 'list' object containing the value 42. The 'push' function frame shows 'self' pointing to the 'Stack instance' and 'item' with the value 42. The 'Return value' is 'None'.