Data Structures

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

Northeastern University

Goals for today

- Review of OOP
- Arrays and pointers
- Linked lists
- Stacks and queues

REVIEW: OOP

What is OOP?

- **Object-oriented programming** (OOP) is a way of *organizing* data and code
- Built around the programming concepts of:
  - **Class** — A definition for a *type* of object
  - **Instance** — A *specific case* of that type of object
  - **Method** — Specialized *functions* for the object

Why use OOP?

- We've already been using OOP!
- Examples of built-in **classes** in Python:
  - Python 2+: `list, tuple, dict, set`
  - Python 3+: `int, float, str`
- In Python 3, *all* data types are classes

Built-in classes in Python

- `list` is a built-in *class* in Python
- `x` is an *instance* of a `list`
- `list.append()` is a *method*

```
list()

x = [1.11, 2.22, 3.33]

x.append(4.0)
```

Class characteristics

- Use `class` keyword to define a *class*
  - Use `def` inside a `class` block to define a *method*
  - Special "magic" methods like `__init__` are *hooks*
  - Access instance *attributes* as `obj.attribute`
- *Good* classes follow OOP principles
  - Encapsulation & abstraction
  - Composition & inheritance
  - Polymorphism

Defining a class in Python

```
class Vector:

    def __init__(self, data):
        self.data = data

    def __str__(self):
        s = ",".join([str(x) for x in self.data])
        return "Vector<" + s + ">"

    def inner(self, y):
        prod = 0
        for xi, yi, in zip(self, y):
            prod += xi * yi
        return prod
```

Defining a class in Python

Class keyword

Name

Initialization method

String (print) method

Method defs

Inner product method

```
class Vector:
    def __init__(self, data):
        self.data = data

    def __str__(self):
        s = ",".join([str(x) for x in self.data])
        return "Vector<" + s + ">"

    def inner(self, y):
        prod = 0
        for xi, yi, in zip(self, y):
            prod += xi * yi
        return prod
```

Defining a class in Python (2)

Class keyword

Name

Refers to instance

"Hook" methods

Access attributes using dot notation

Regular method

```python
class Vector:

    def __init__(self, data):
        self.data = data

    def __str__(self):
        s = ",".join([str(x) for x in self.data])
        return "Vector<" + s + ">"

    def inner(self, y):
        prod = 0
        for xi, yi, in zip(self, y):
            prod += xi * yi
        return prod
```

Defining methods in Python

- Function `def` inside `class` block creates a method
- First argument to a method should be `self`
  - Use `self` as a handle to the specific instance of the class
  - In practice, `foo(self, arg)` is called as `obj.foo(arg)`
- Special "magic" methods are hooks into Python
  - `__init__` is used to initialize instances of the class
  - `__add__` and `__mul__` implement `+` and `*`, etc.

Methods and `self`

- First argument to methods should be `self`
- Python passes the object as the first argument
  - Similar to `this` keyword in other languages like Java or C++
  - Passing of instance is *explicit* rather than *implicit* in Python
  - Use of "self" name is a (strong) convention, not a keyword
- Use as a handle to the "current" instance

Accessing attributes

- Access instance attributes using `obj.attribute`
  - Usually `self.attribute` inside a method
  - No private attributes — users can access them too!
- Get or set data attributes of an object
  - Typically, use `__init__` method to set initial values of attributes
  - Other methods may be used to change values of instance attributes
- Objects are *mutable* by default

"Magic" methods

- Nothing "magic" about double-underscore methods
  - "Dunder" methods are a core part of OOP system in Python
  - Use to make user-defined classes behave like built-in classes
- Special methods are hooks into Python operators
  - `__init__` is used to initialize instances of the class
  - `__add__` and `__mul__` implement `+` and `*`, etc.
- Only *really* need to know `__init__` for basic use

Special methods

Method	Implements
__init__	Object initialization
__repr__	print()
__str__	str()
__len__	len()
__iter__	iter()
__next__	next()
__reversed__	reversed()
__contains__	value in self

Method	Implements
__add__	self + value
__sub__	self - value
__mul__	self * value
__eq__	self == value
__lt__	self > value
__and__	self and value
__or__	self or value
__getitem__	self[i]

. . . and many more!

Inheritance in Python

- Classes can *inherit* from super-classes
  - Enclose names of super-classes in parentheses after class name
  - E.g., `class SubName(SuperName)`
- Methods are inherited from the super-classes
  - Overwrite methods by re-defining them in sub-class
  - Use `super()` to access a *proxy instance* of the super-class to use the super-class versions of re-defined methods
- Possible to inherit from multiple classes

Defining a sub-class in Python

```python
class Person:

    def __init__(self, name = "Jane Smith", uid = "000000"):
        self.name = name
        self.uid = uid

    def __str__(self):
        cls = self.__class__.__name__
        return "{}(name: {}, uid = {})".format(cls, self.name, self.uid)

class Employee(Person):

    def __init__(self, title = None, salary = 0, **kwargs):
        super().__init__(**kwargs)
        self.title = title
        self.salary = salary
```

Defining a sub-class in Python

Super-class of `Employee`

Default params

Default params

```
class Person:

    def __init__(self, name = "Jane Smith", uid = "000000"):
        self.name = name
        self.uid = uid

    def __str__(self):
        cls = self.__class__.__name__
        return "{}(name: {}, uid = {})".format(cls, self.name, self.uid)
```

Sub-class of `Person`

Packed `dict` of keyword args

```
class Employee(Person):

    def __init__(self, title = None, salary = 0, **kwargs):
        super().__init__(**kwargs)
        self.title = title
        self.salary = salary
```

Super-class proxy

Pass unpacked `dict` of keyword args

Instances and attributes

- Construct an **instance** of a class
  - `x = Vector([1, 2, 3])`
- Access attributes using dot notation
  - Data attributes: `x.data`
  - Methods: `x.inner(y)`

Information hiding

- *Anyone* can change data attributes
- Public access to data can cause problems:
  - Change data in a way that results in invalid object state
  - Implementation (internal attribute names) may change
  - Leads to errors and hard-to-find bugs
- Avoid public access of data attributes
- Better to use **getter** and **setter** methods

Getters and setters

```
class Person:

    def __init__(self, name = "Jane Smith", uid = "000000"):
        self.name = name
        self.uid = uid

    def getname(self):
        return self.name

    def setname(self, name):
        self.name = name

    def getuid(self):
        return self.uid

    def setuid(self, uid):
        self.uid = uid
```

Getters and setters

```
class Person:

    def __init__(self, name = "Jane Smith", uid = "000000"):
        self.name = name
        self.uid = uid

    def getname(self):
        return self.name
```

Get the value of `obj.name`

```
    def setname(self, name):
        self.name = name
```

Set the value of `obj.name`

```
    def getuid(self):
        return self.uid
```

Get the value of `obj.uid`

```
    def setuid(self, uid):
        self.uid = uid
```

Set the value of `obj.uid`

Getter and setter methods

- Methods dedicated to accessing public data
- Encourage use of getter/setter methods
  - Code still works even if internal implementation changes!
  - Easier to maintain and reason over code
- "Private" data won't have getter/setter
  - No way to force private attributes in Python—rely on best practice
  - A single leading underscore (e.g., `_var`) "hints" an attribute is private

Power of OOP

- Encapsulation
  - Bundle data and methods while hiding implementation details
- Composition
  - Objects may *contain* other objects to make complex objects
- Inheritance
  - Child classes may *inherit* behavior from their parent classes
- Polymorphism
  - Many data types can share a common interface

DATA STRUCTURES

Data structures

- Programs need to store data
- Best way to store data *depends on* ***how*** *data*:
  - Is **written** to the data structure
  - Is **read** from the data structure
  - Is **modified** in the data structure
- Consider needs of a program when choosing the most appropriate data structure

Abstract vs. concrete

- **Abstract** data type:
  - *Define characteristics and operations* for the data structure
  - May not guarantee any performance requirements
- **Concrete** data type:
  - The data structure is *defined by its implementation*
  - Has specific performance measurements
- An *abstract data type* may be implemented using more than one *concrete data types*

Review: Objects

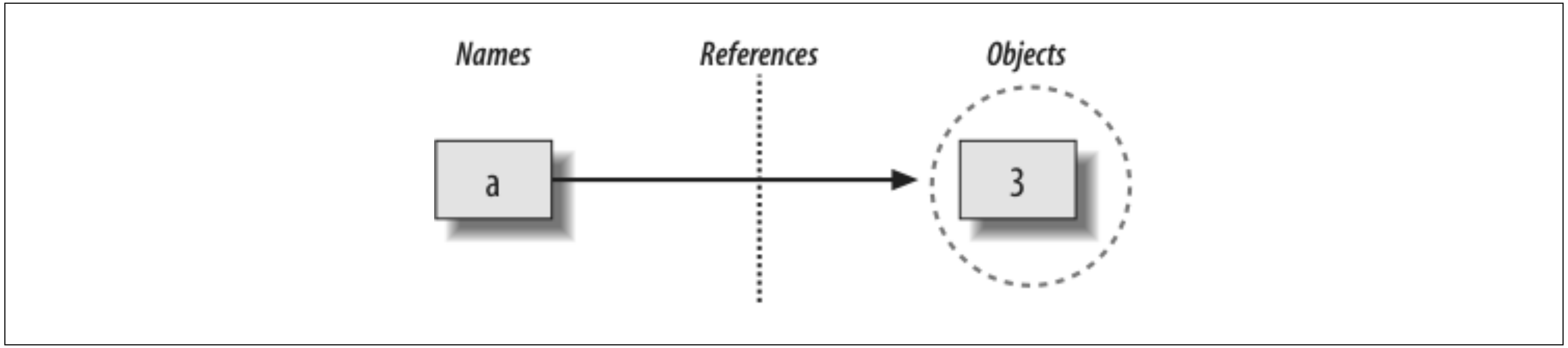
- Programs manipulate **objects**
- Objects are the "things" that exist in a program
- Objects:
  - Are stored in **memory** with **value**(s) associated with them
  - Have a **data type** that defines what **operations** can be performed
  - Are frequently bound to **variable** names that identify them

Review: Variables

- Programs refer to **variables**
- A variable consists of:
  - Storage location in memory
  - Name
  - Value (a specific object)
- **Assignment** binds a **value** to a variable **name**

Variables create **references**

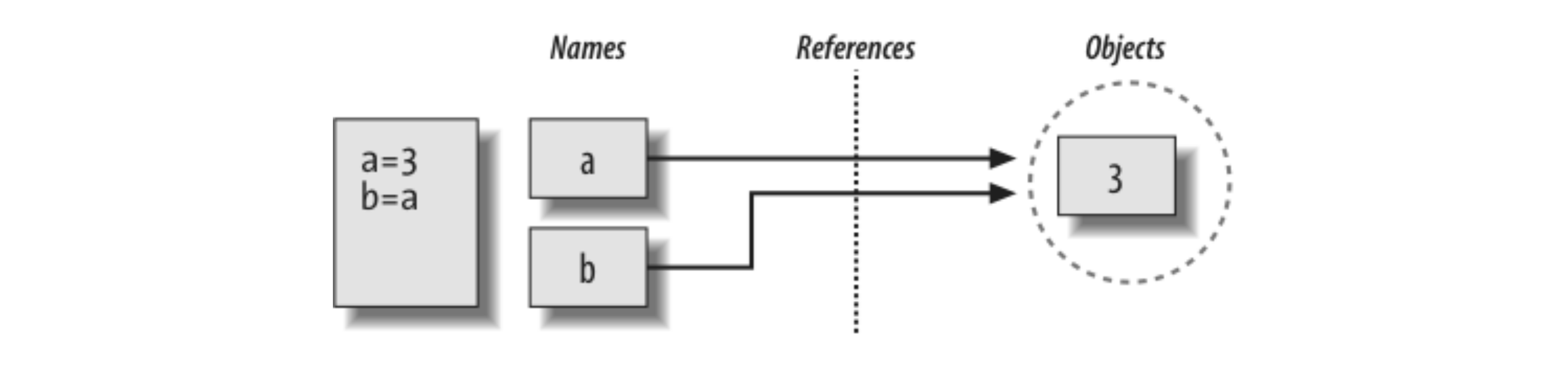
- Link between variable name and object
  - This link is called a ***reference***
  - An object may have multiple references
- Variables *point* to an object in memory



Learning Python. Mark Lutz. O'Reilly Media, 2013.

Pointers and shared references

- Multiple variables may reference the same object
  - Multiple variables may *point* to same location in memory
  - But only a single version of the object exists
- No additional memory is used

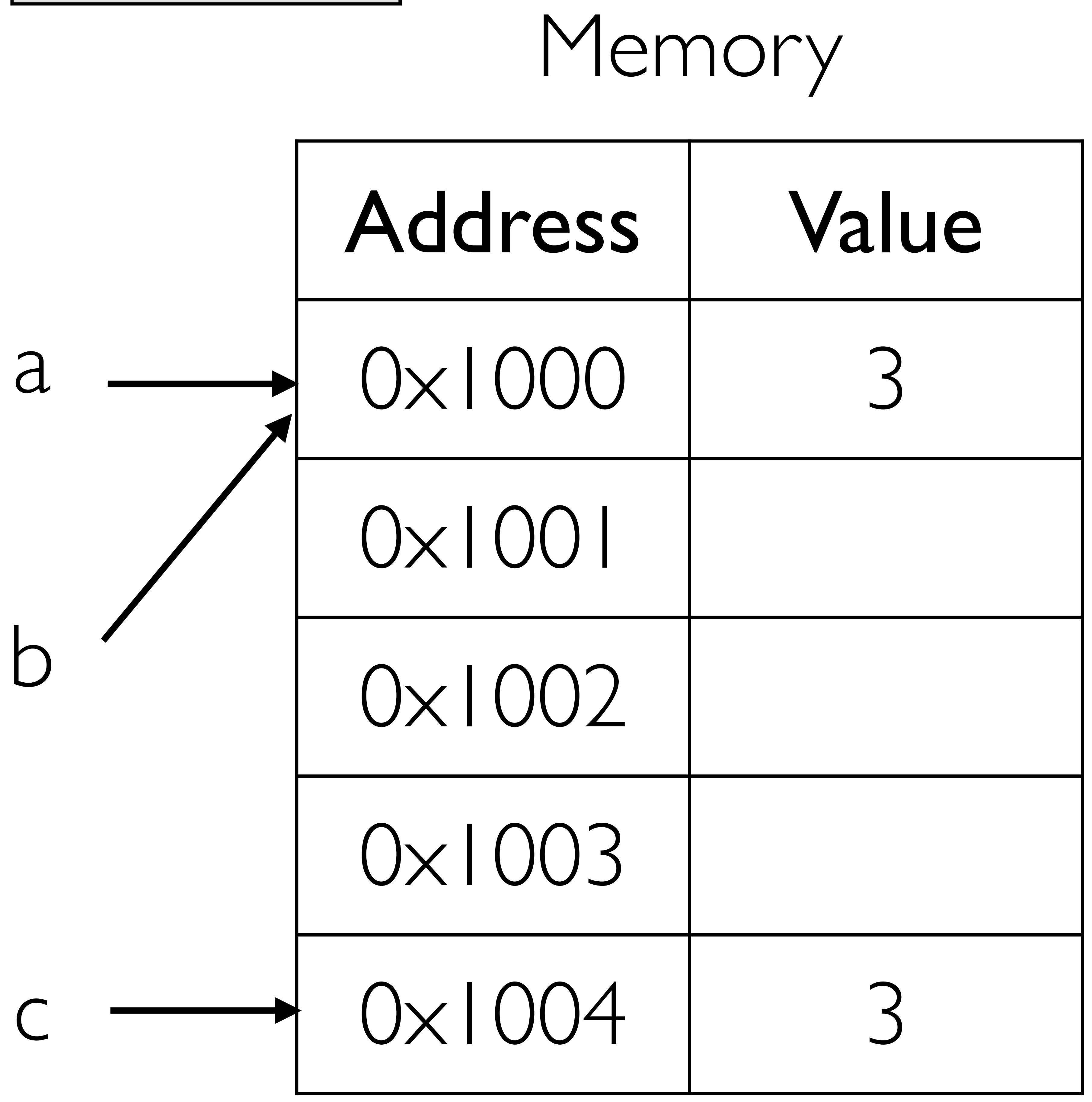


Learning Python. Mark Lutz. O'Reilly Media, 2013.

Pointers and memory

```
>>> a = 3
>>> b = a
>>> c = 3
```

- Variables point to a *location* in **memory** where an object is stored
- Different *types* of objects require different **space** in memory
- E.g., a **double** *float* is typically 64-bit (i.e., requires 8 bytes in memory)

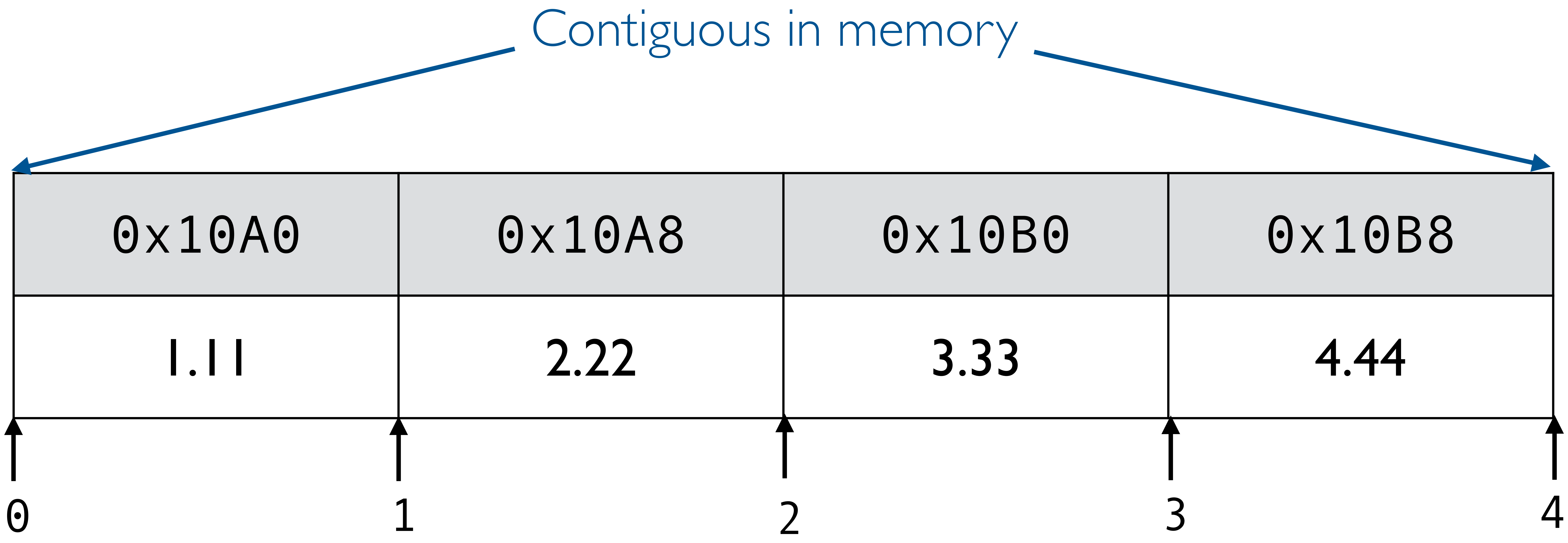


Arrays

- **Ordered sequence** data type
- Stored in a single *block of memory*
- Items are stored ***contiguously*** in memory
- All items must be the ***same data type***

Arrays in Python

```
import array as arr

x = arr.array("d", [1.11, 2.22, 3.33, 4.44])
```



Quick access to items by offset

Performance of arrays

- Very fast random read/write of existing items
- Very fast traversal of items (contiguous in memory)
- Somewhat slow searching for specific items
- Very slow insertion/deletion of new items

Appending to an array in Python

```
x = arr.array("d", [1.11, 2.22, 3.33, 4.44])

x.append(5)
```

Need to allocate a new block of memory

Old

0x10A0	0x10A8	0x10B0	0x10B8
1.11	2.22	3.33	4.44

Copy
old elements
to new array

New

0x20A0	0x20A8	0x20B0	0x20B8	0x20C0
1.11	2.22	3.33	4.44	5.0

Considerations for data structures

- What performance characteristics are needed?
  - **Read/write** (of existing items)
  - **Insertion/deletion** (of new items)
  - **Traversal** (iteration over all items)
  - **Searching** (find a specific item)
- Memory space requirements

LINKED LISTS

The cons cell

- "Cons" cell is a flexible building block
- Consider a simple data structure
  - A simple 2-tuple storing two items
  - Each item is a *value* or a *pointer* to another cons cell
- Can be nested to **cons**truct powerful recursive data structures

A cons cell class in Python

```
class CONS:

    def __init__(self, first=None, rest=None):
        self.first = first
        self.rest = rest

    def getfirst(self):
        return self.first

    def getrest(self):
        return self.rest

    def setfirst(self, first=None):
        self.first = first

    def setrest(self, rest=None):
        self.rest = rest
```

A cons cell class in Python

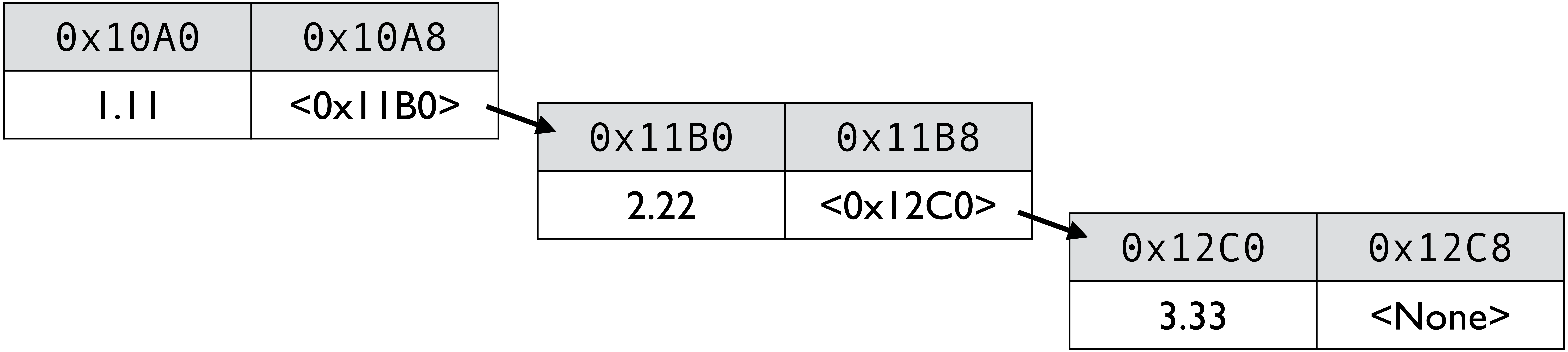
```python
class CONS:

    def __init__(self, first=None, rest=None):
        self.first = first
        self.rest = rest

    def getfirst(self):
        return self.first

    def getrest(self):
        return self.rest

    def setfirst(self, first=None):
        self.first = first

    def setrest(self, rest=None):
        self.rest = rest
```

Getter methods

Setter methods

Linked list made of cons cell

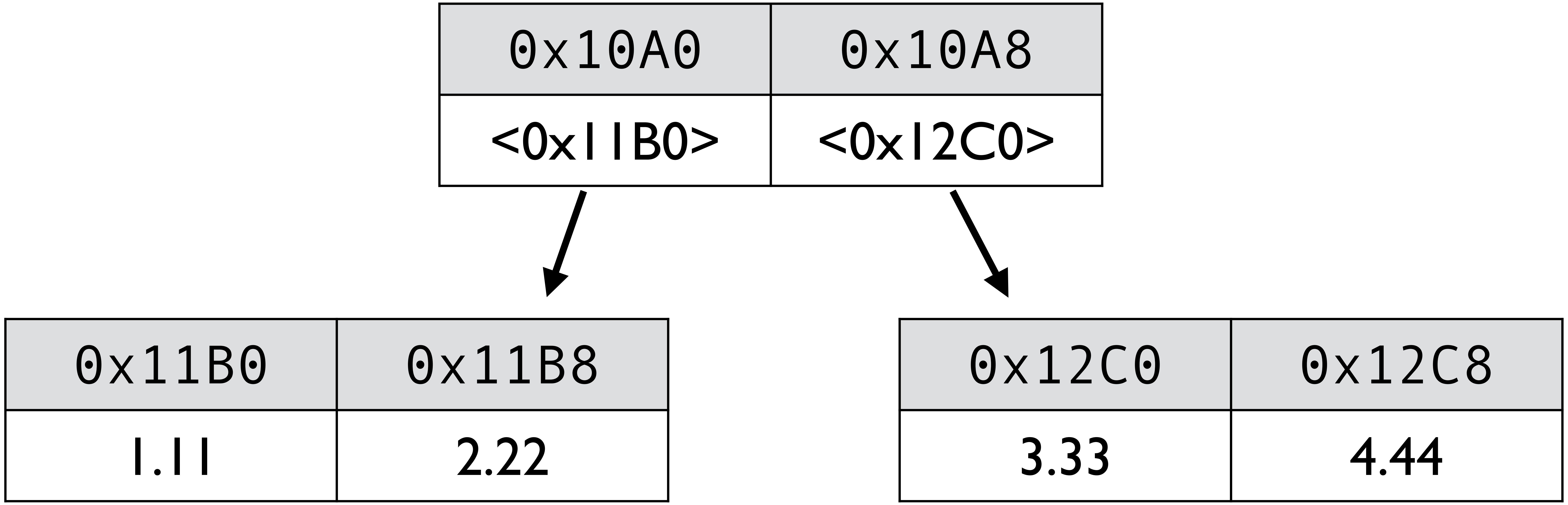
```
(1.11 · (2.22 · (3.33 · None)))
```



The cells in a linked list are typically called ***nodes***

Binary tree made of cons cell

```
((1.11 · 2.22) · (3.33 · 4.44))
```



More on trees another time

Beyond cons cells

- The **cons cell** is an important idea in *recursive* data structures
- Used extensively in functional languages (especially LISP-like languages)
- Can be used to build lists and trees
- Some *linked list* variants require nodes more complex than cons cells

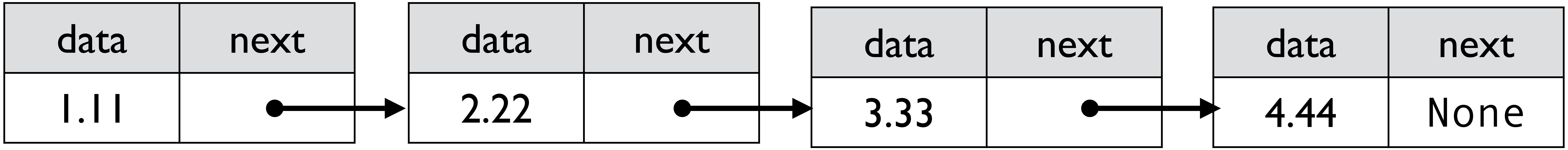
Linked lists

- **Ordered sequence** data type
- Items stored in *linked nodes*
- Nodes stored ***non-contiguously*** in memory
- Data types of the items is not specified
  - Homogeneity or heterogeneity depends on implementation

Singly-linked lists

- Linked lists are a chain of nodes
- Each node stores data and points to next node

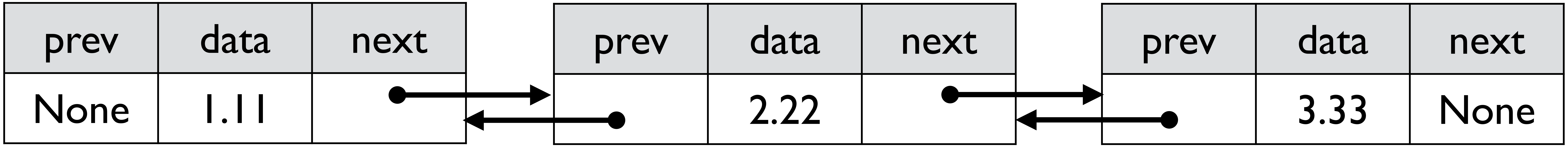
```
(1.11 · (2.22 · (3.33 · (4.44 · None))))
```



Nodes are typically *not* contiguous in memory

Doubly-linked lists

- Nodes also point to the *previous* node
- Traverse list in either direction
- Link first and last node to make list *circular*



Uses additional memory for increased flexibility

Linked lists in Python

```
class LList:

    def __init__(self):
        self.head = None

    def append(self, value):
        newcell = CONS(value, None)
        if self.head is None:
            self.head = newcell
        else:
            tail = self.head
            while tail.getrest() is not None:
                tail = tail.getrest()
            tail.setrest(newcell)
```

Linked lists in Python

```
class LList:

    def __init__(self):
        self.head = None

    def append(self, value):
        newcell = CONS(value, None)
        if self.head is None:
            self.head = newcell
        else:
            tail = self.head
            while tail.getrest() is not None:
                tail = tail.getrest()
            tail.setrest(newcell)
```

Nodes are cons cells

Traverse list to append item at end

Performance of linked lists

- Very slow random read/write of existing items
- Fast traversal of items (non-contiguous in memory)
- Somewhat slow searching for specific items
- Fast insertion/deletion of new items
    - Depends on location in the list

Linked list vs. array

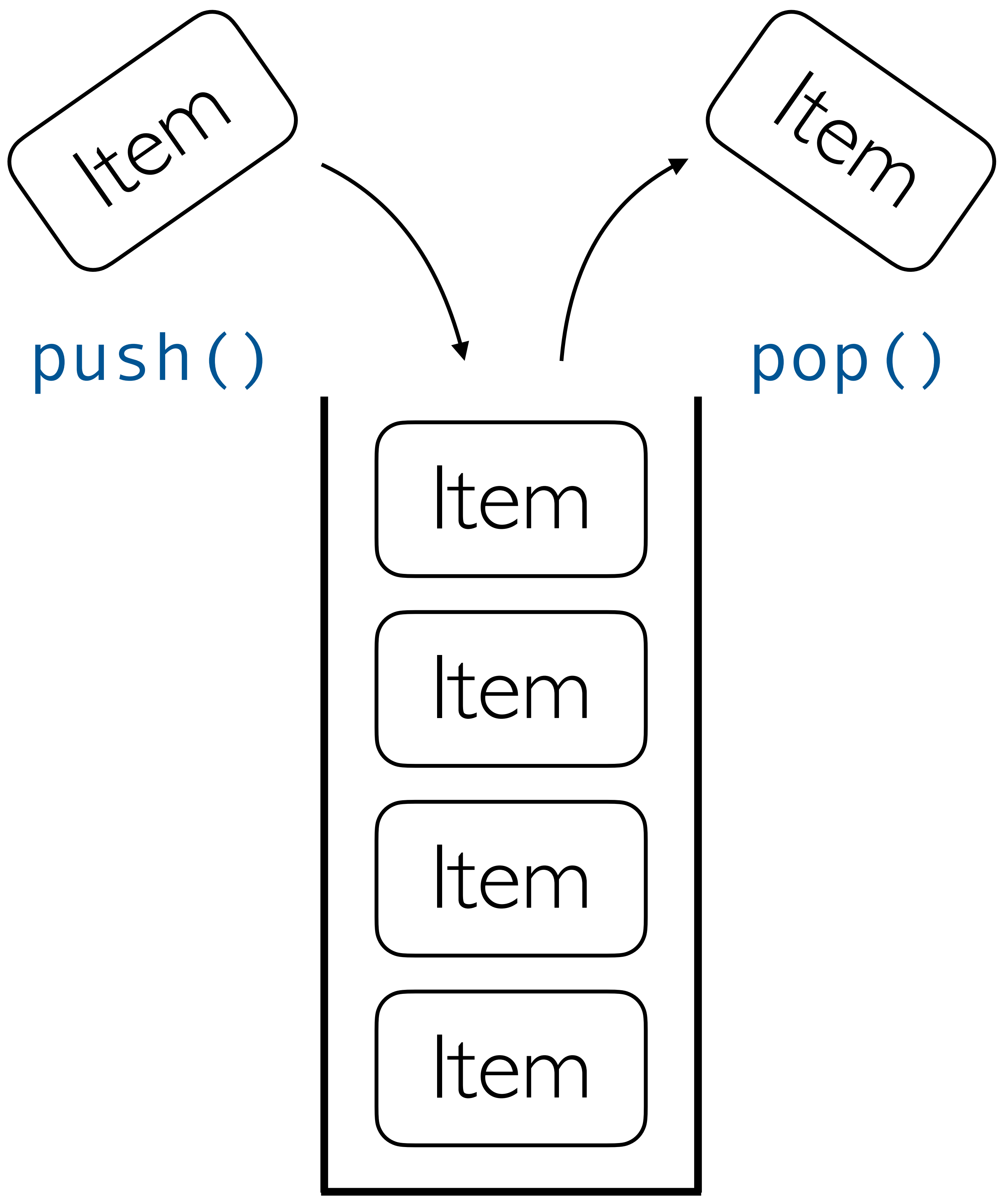
Array	Linked List
Contiguous in memory	Non-contiguous in memory
Homogenous data types	Heterogenous data types
Fast random access	Slow random access
Slow append/insert/delete	Fast append/insert/delete

STACKS AND QUEUES

Stacks

- *Abstract* **ordered sequence** data type
- Must add/remove items in order
- Last-in, first-out (LIFO)

Stack characteristics

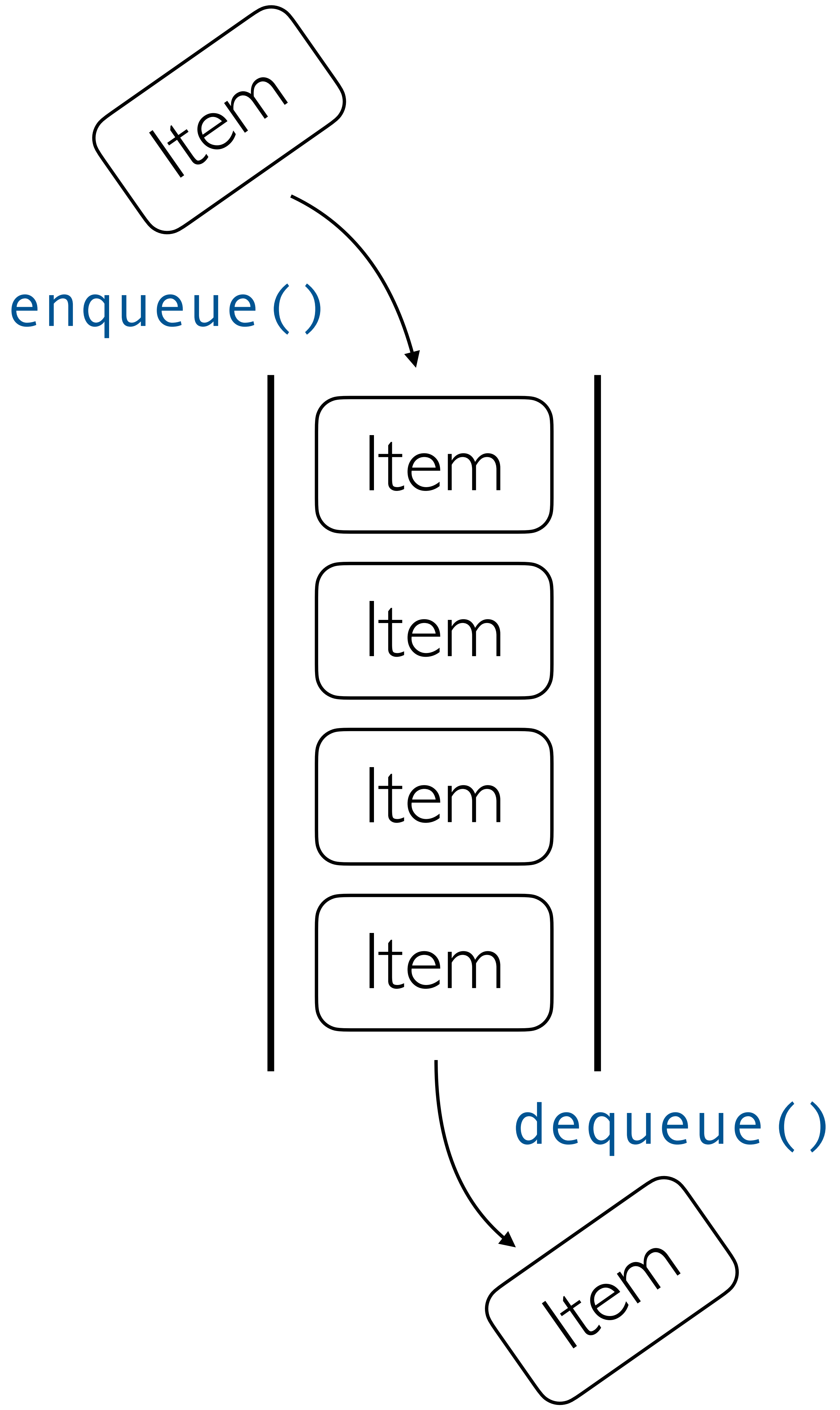


- Last-in, first-out (LIFO)
- Two primary operations:
  - **Push**: add item to top of stack
  - **Pop**: remove *and* return the top-most item of the stack
- Cannot access middle elements

Queues

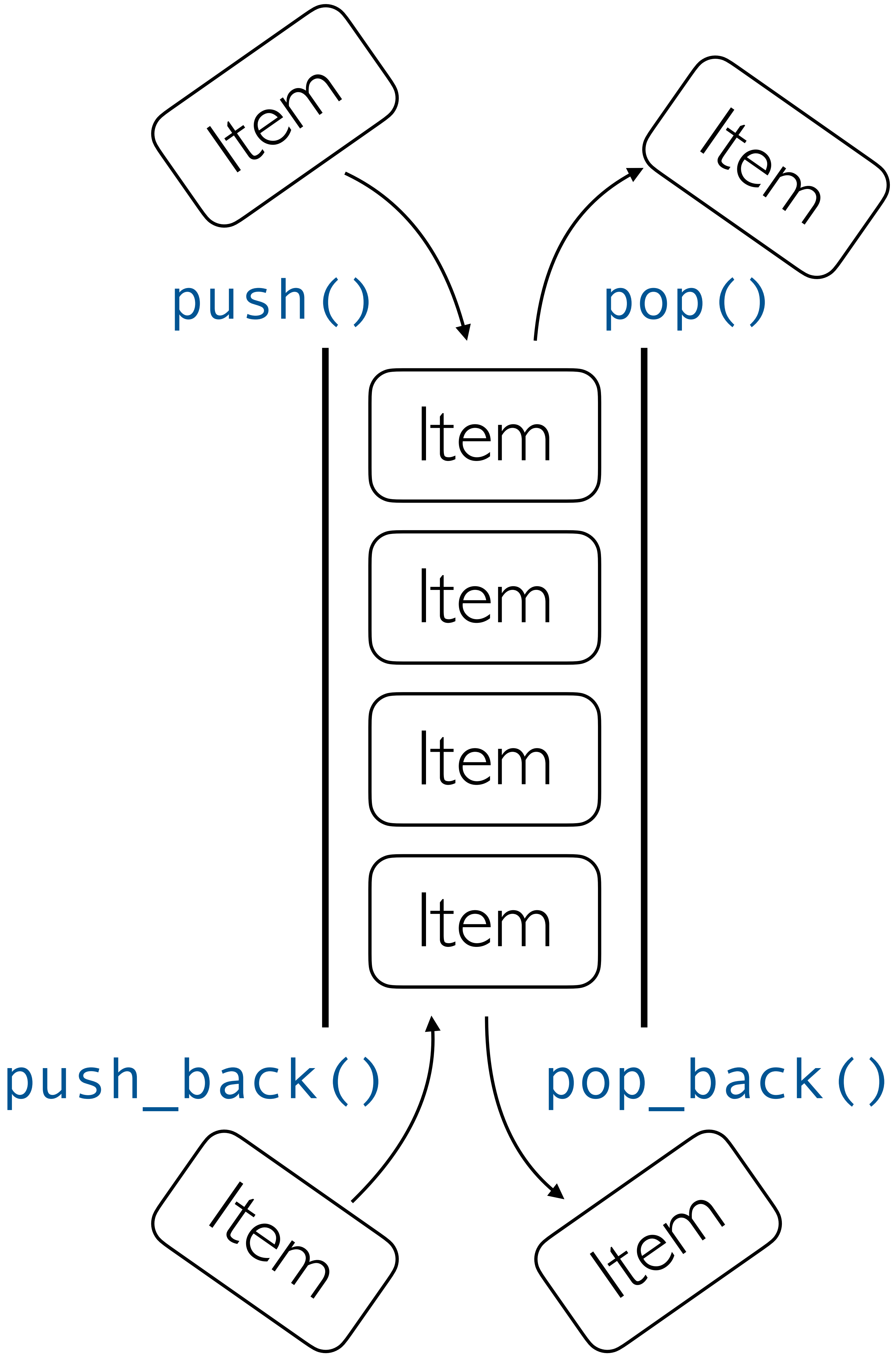
- *Abstract* **ordered sequence** data type
- Must add/remove items in order
- First-in, first-out (FIFO)

Queue characteristics



- First-in, first-out (FIFO)
- Two primary operations:
    - **Enqueue**: add item to end of queue
    - **Dequeue**: remove *and* return item from the front of queue
- Cannot access middle elements

Deque characteristics



- *Double-ended* queue
- Four primary operations:
  - **Push & push_back**: add item to front/end of the deque
  - **Pop & pop_back**: remove *and* return front/end of deque
- Cannot access middle elements

Stacks and queues

- Many practical applications in computer science
- Stacks
  - Function calls go on the *call stack*
  - Parsing of language expressions
  - Memory management (allocating + freeing)
- Queues
  - CPU and I/O scheduling
  - Data traffic over a network
  - Algorithms such as breadth-first search (BFS)

More on stacks and queues

- Examples of *abstract* data structures
- Could be implemented using:
  - Arrays
  - Linked lists
- What are the advantages/disadvantages of using arrays vs. linked lists for:
  - A stack?
  - A queue?

Lists in Python

- How are lists in implemented in Python?

Lists in Python

- Built-in lists in Python are *arrays of pointers*
- Good compromise of performance vs. flexibility