Lists

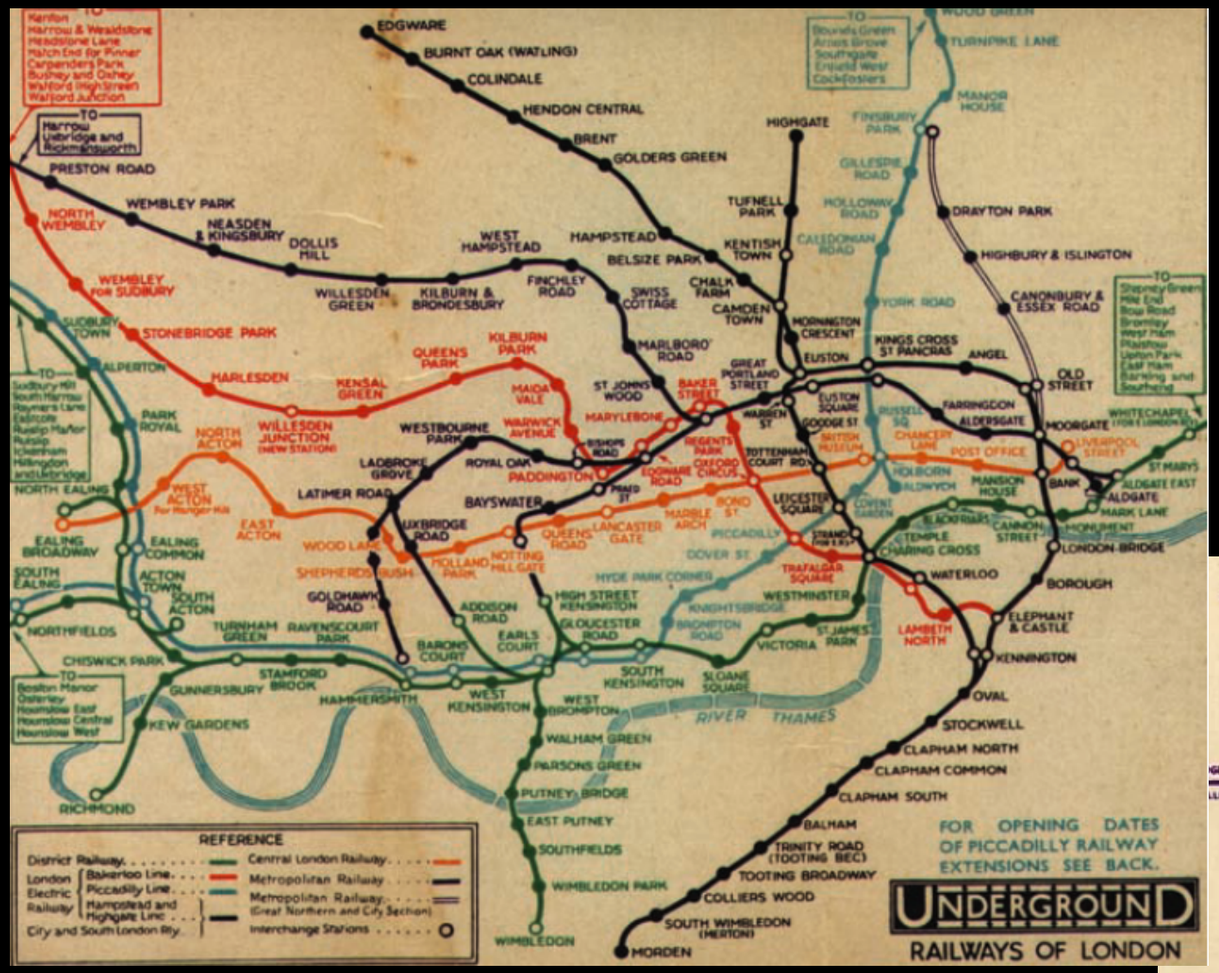
CAC 210

Amber Wagner
Birmingham-Southern College

How does print() work?

Abstraction

Abstraction



1928

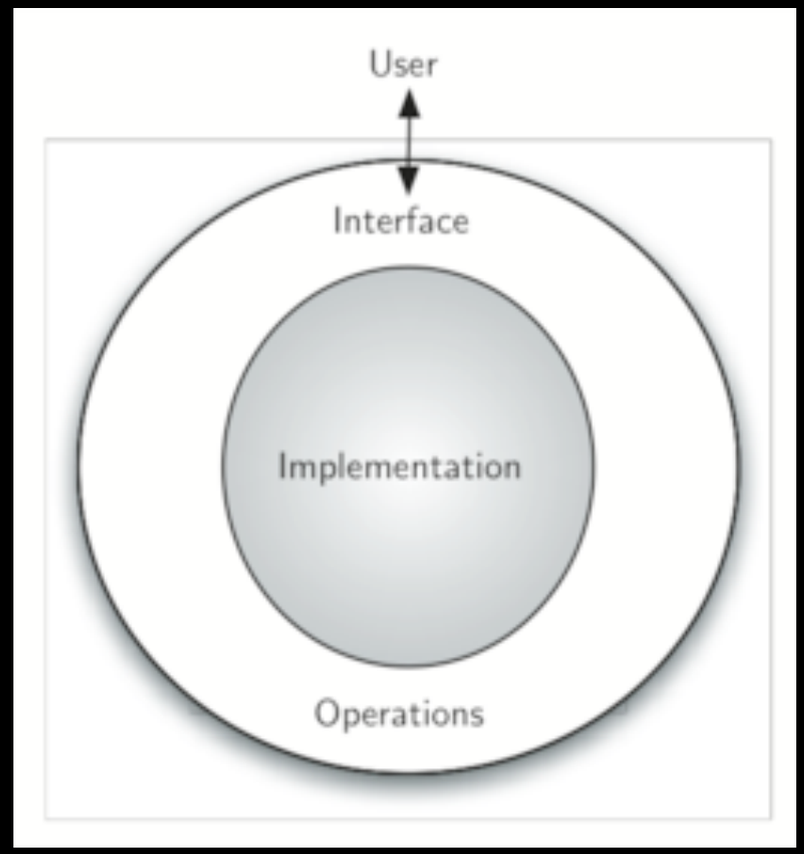
1933

Abstraction

- The print() method was written to allow Python programmers to focus on the bigger picture
- The print() method abstracts away the details of how print() is actually implemented
- This is called procedural abstraction

Data Abstraction

- Logical description of how we think about the data including operations for interacting with the data



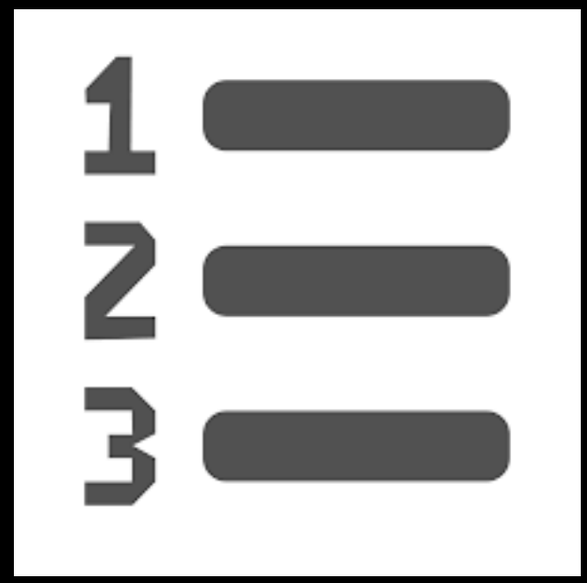
- No concern with implementation of the data type nor the operations
- Only concerned with what data represents
- Idea behind encapsulation
- We are going to concern ourselves with the structure today

http://interactivepython.org/runestone/static/pythonds/Introduction/WhyStudyDataStructuresandAbstractDataTypes.html

Data Structure

- An implementation of an abstract data type is referred to as a data structure
- How long an algorithm takes often depends on how the data is organized
- Different data structures will give different running times
- It's important to use the appropriate data structure for your task
- Let's take a look at lists…

Lists



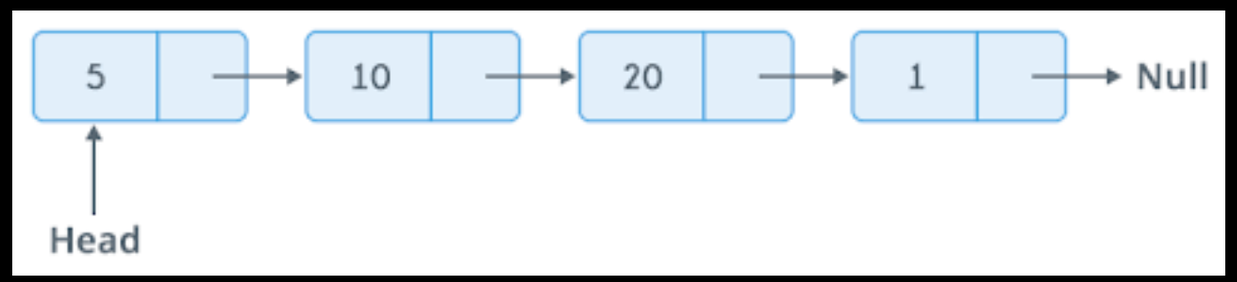
- Abstract Data Type
- Holds ordered data
- Linear structure
- Each element within a list can be thought of as a node

Lists: Common Methods

Operation	Description	Example starting with list: 99, 77
Append(list, x)	Inserts x at end of list	Append(list, 44), list: 99, 77, 44
Prepend(list, x)	Inserts x at start of list	Prepend(list, 44), list: 44, 99, 77
InsertAfter(list, w, x)	Inserts x after w	InsertAfter(list, 99, 44), list: 99, 44, 77
Remove(list, x)	Removes x	Remove(list, 77), list: 99
Search(list, x)	Returns item if found, else returns null	Search(list, 99), returns item 99 Search(list, 22), returns null
Print(list)	Prints list's items in order	Print(list) outputs: 99, 77
PrintReverse(list)	Prints list's items in reverse order	PrintReverse(list) outputs: 77, 99
Sort(list)	Sorts the lists items in ascending order	list becomes: 77, 99
IsEmpty(list)	Returns true if list has no items	For list 99, 77, IsEmpty(list) returns false
GetLength(list)	Returns the number of items in the list	GetLength(list) returns 2

Singly-Linked List

- The variable referenced by the list's name stores a pointer to the head and the tail of the list
- Each node within the list has a value and a pointer to the next element in the list
- When adding or removing elements, one has to be careful not to drop the pointer



https://visualgo.net/en/list

Singly-Linked List

```
ListAppend(list, newNode) {
  if (list->head == null) { // List empty
     list->head = newNode
     list->tail = newNode
  }
  else{
     list->tail->next = newNode
     list->tail = newNode
  }
}
```

```
ListPrepend(list, newNode) {
  if (list->head == null) { // list empty
     list->head = newNode
     list->tail = newNode
  }
  else {
     newNode->next = list->head
     list->head = newNode
  }
}
```

Singly-Linked List

```
ListInsertAfter(list, curNode, newNode) {
   if (list->head == null) { // List empty
      list->head = newNode
      list->tail = newNode
   }
   else if (curNode == list->tail) { // Insert after tail
      list->tail->next = newNode
      list->tail = newNode
   }
   else {
      newNode->next = curNode->next
      curNode->next = newNode
   }
}
```

Singly-Linked List

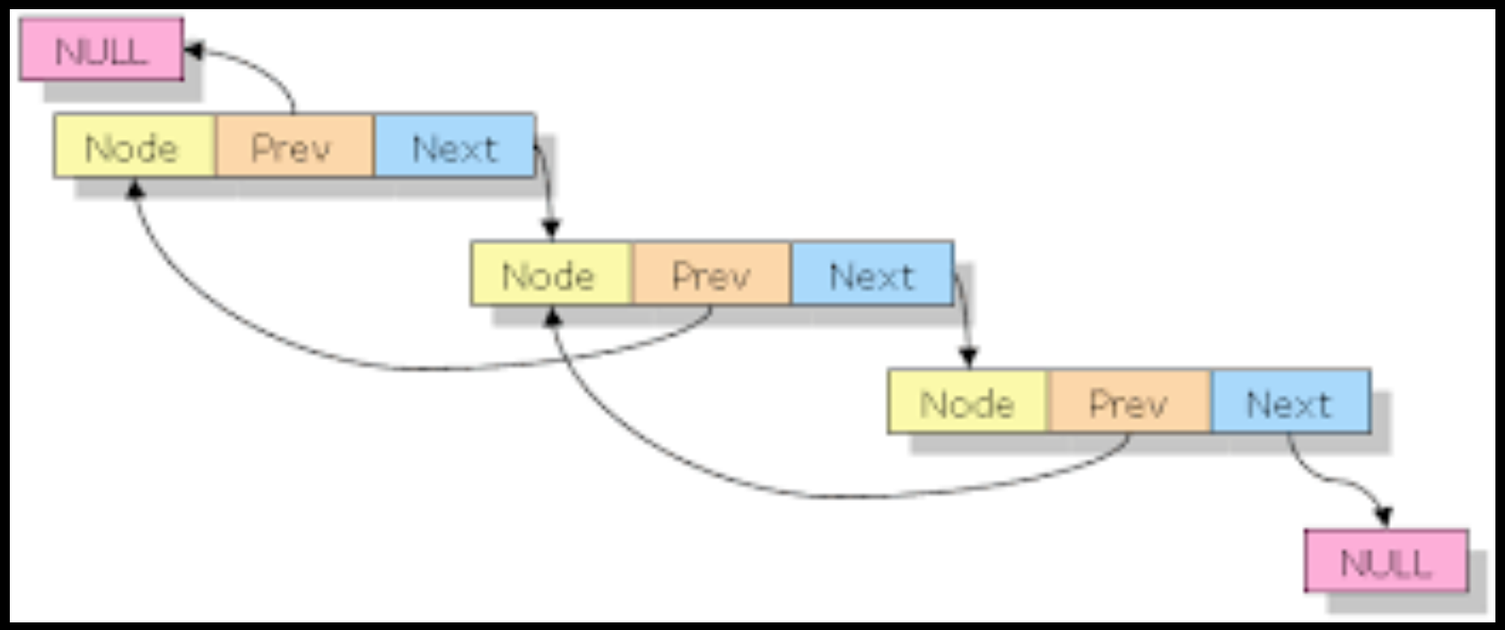
```
ListRemoveAfter(list, curNode) {
  // Special case, remove head
  if (curNode is 0 && list->head is not null) {
     sucNode = list->head->next
     list->head = sucNode

     if (sucNode is null) { // Removed last item
        list->tail = null
     }
  }
  else if (curNode->next is not null) {
     sucNode = curNode->next->next
     curNode->next = sucNode

     if (sucNode is null) { // Removed tail
        list->tail = curNode
     }
  }
}
```

Doubly-Linked List

- Similar to a singly-linked list, but it has a pointer to the previous and the next nodes



Doubly-Linked Lists

```
ListAppend(list, newNode) {
  if (list->head == null) { // List empty
     list->head = newNode
     list->tail = newNode
  }
  else {
     list->tail->next = newNode
     newNode->prev = list->tail
     list->tail = newNode
  }
}
```

```
ListPrepend(list, newNode) {
  if (list->head == null) { // List empty
     list->head = newNode
     list->tail = newNode
  }
  else {
     newNode->next = list->head
     list->head->prev = newNode
     list->head = newNode
  }
}
```

Doubly-Linked Lists

```
ListInsertAfter(list, curNode, newNode) {
  if (list->head == null) { // List empty
     list->head = newNode
     list->tail = newNode
  }
  else if (curNode == list->tail) { // Insert after tail
     list->tail->next = newNode
     newNode->prev = list->tail
     list->tail = newNode
  }
  else {
     sucNode = curNode->next
     newNode->next = sucNode
     newNode->prev = curNode
     curNode->next = newNode
     sucNode->prev = newNode
  }
}
```

Doubly-Linked Lists

```
ListRemove(list, curNode) {
   sucNode = curNode->next
   predNode = curNode->prev

   if (sucNode is not null) {
      sucNode->prev = predNode
   }

   if (predNode is not null) {
      predNode->next = sucNode
   }

   if (curNode == list->head) { // Removed head
      list->head = sucNode
   }

   if (curNode == list->tail) { // Removed tail
      list->tail = predNode
   }
}
```