python™

Python in 90 minutes

Joe James

Math & Computer Science Tutorials
from Silicon Valley

100% BS

Python Variables

Built-in Types

Integer

Floating point

String

Boolean

Complex number

Type Conversion

int()	# string to integer
float()	# string to float
str()	# number to string
bool()	# 0, [], None => False
hex()	# decimal to hex
ord()	# ASCII value

Variable Naming Tips

- Naming can have letters, numbers and underscore, but cannot start with a number
- Some Python reserved words cannot be used (eg. if, for, in, open)
- Use descriptive variable names
  - first_name, date_of_birth, hair_color
- Case Matters
  - name is not the same as Name
- Constants in all caps:
  PI = 3.14159, DOZEN = 12

All variables in Python are *reference variables*, meaning the variable contains a memory address to where the data is stored.

	Memory Address	**Data**
name →	21F7h	‘Cassandra’

In Python, most variables are ***immutable***, meaning they don’t change in-place.

Python creates a new value in a different memory location when a variable changes.

Integer	Immutable
Float	Immutable
String	Immutable
Tuple	Immutable
List	Mutable
Set	Mutable
Dictionary	Mutable

bool(expression)

Boolean: True or False

Evaluate to FALSE

0

0.0

""

[]

None

1 and 0

0 or ""

5 - 5

3 < 2

-1 > 33

8 >= 100

5 <= 1

0 == 88

1.2 != 1.2

Evaluate to TRUE

any non-zero number

any non-empty string

any non-empty list

1

1 or 0

81 and -23

‘pig’

‘cat’ == ‘cat’

[‘dog’]

‘a’ < ‘b’

10 > 5

-1 < 33

8 >= 8

0 == 0

1.2 != 1.3

5 > 3 and 10

1 == 0 or [0]

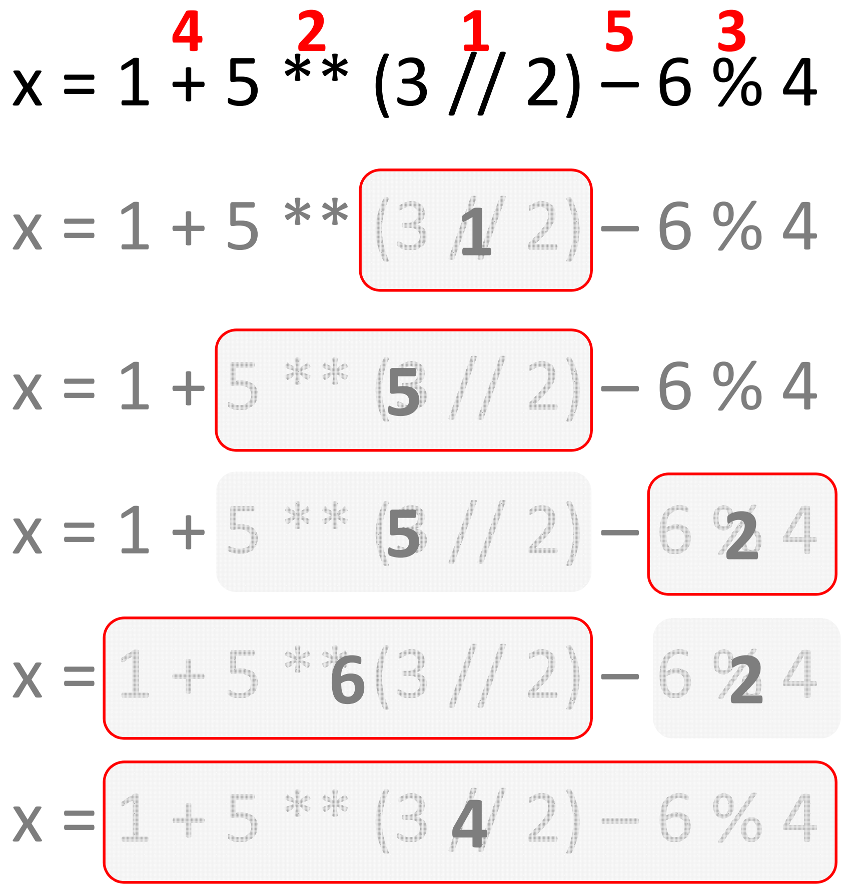
Python Math functions

Symbol	Function	Example	Result
+	addition	5 + 3	8
–	subtraction	10 – 6	4
*	multiplication	3 * 7	21
//	integer division	15 // 6	2
/	float division	15 / 6	2.5
**	power	7 ** 2	49

Order of Operations

1. ( )
2. **
3. * / // %
4. + –
5. left to right

Example:



x = 1 + 5 ** (3 // 2) – 6 % 4

x = 1 + 5 ** $\boxed{1}$ – 6 % 4

x = 1 + $\boxed{5}$ – 6 % 4

x = 1 + 5 – $\boxed{2}$

x = $\boxed{6}$ – 2

x = $\boxed{4}$

Indexing Starts with 0

String: “Vikash”

List: [‘V’, ‘i’, ‘k’, ‘a’, ‘s’, ‘h’]

Index	**Value**
0	V
1	i
2	k
3	a
4	s
5	h

String *Escape Sequences*

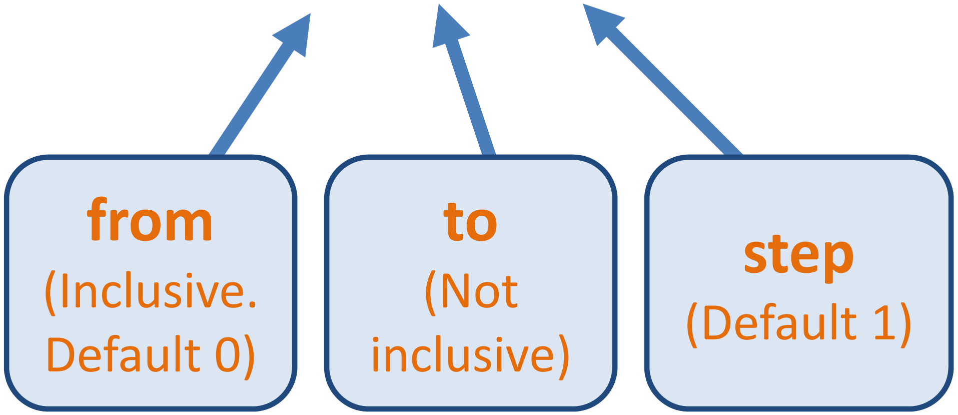
\n	newline	print(‘my\ndog’)	my dog
\t	tab	print(‘my\tcat’)	my cat
\	backslash	print(‘my\\turtle’)	my\turtle

range

range gives a sequence of integers

“to” means up to but not including.

range(2, 10, 3) **# returns [2, 5, 8]**



from
(Inclusive.
Default 0)

to
(Not
inclusive)

step
(Default 1)

for i in range(5):	[0, 1, 2, 3, 4]
for i in range(7, 9):	[7, 8]
for i in range(1, 10, 2):	[1, 3, 5, 7, 9]
for i in range(10, 6, -1):	[10, 9, 8, 7]

List

General purpose

Most widely used data structure

Grow and shrink size as needed

Sequence type

Sortable

Tuple

Immutable (can’t add/change)

Useful for fixed data

Faster than Lists

Sequence type

Set

Store non-duplicate items

Very fast access vs Lists

Math Set ops (union, intersect)

Unordered

Dict

Key/Value pairs

Associative array, like Java HashMap

Unordered

SEQUENCES (String, List, Tuple)

- indexing: x[6]
- slicing: x[1:4]
- adding/concatenating: +
- multiplying: *
- checking membership: in/not in
- iterating for i in x:
- len(sequence1)
- min(sequence1)
- max(sequence1)
- sum(sequence1[1:3]])
- sorted(list1)
- sequence1.count(item)
- sequence1.index(item)

SEQUENCES

String List Tuple

- **indexing**
  - Access any item in the sequence using its index

String

```
x = 'frog'
print (x[3])                        # prints 'g'
```

List

```
x = ['pig', 'cow', 'horse']
print (x[1])                        # prints 'cow'
```

- **slicing**
  - Slice out substrings, sublists, subtuples using indexes

  [start : end : step]

```
x = 'computer'
```

Code	Result	Explanation
`x[1:4]`	'omp'	Items 1 to 3
`x[1:6:2]`	'opt'	Items 1, 3, 5
`x[3:]`	'puter'	Items 3 to end
`x[:5]`	'compu'	Items 0 to 4
`x[-1]`	'r'	Last item
`x[-3:]`	'ter'	Last 3 items
`x[:-2]`	'comput'	All except last 2 items

- **adding / concatenating**
  - Combine 2 sequences of the same type using **+**

String

```
x = 'horse' + 'shoe'
print (x)              # prints 'horseshoe'
```

List

```
x = ['pig', 'cow'] + ['horse']
print (x)          # prints ['pig', 'cow', 'horse']
```

- **multiplying**
  - Multiply a sequence using *****

String

```
x = ‘bug' * 3
print (x)            # prints ‘bugbugbug'
```

List

```
x = [8, 5] * 3
print (x)            # prints [8, 5, 8, 5, 8, 5]
```

- **checking membership**
  - Test whether an item is **in** or **not in** a sequence

String

```
x = 'bug'
print ('u' in x)                 # prints True
```

List

```
x = ['pig', 'cow', 'horse']
print ('cow' not in x)           # prints False
```

- **iterating**
  - Iterate through the items in a sequence

Item

```
x = [7, 8, 3]
for item in x:
    print (item * 2)                 # prints 14, 16, 6
```

Index & Item

```
x = [7, 8, 3]
for index, item in enumerate(x):
    print (index, item)              # prints 0 7, 1 8, 2 3
```

- **number of items**
  - Count the number of items in a sequence

String

```
x = 'bug'
print (len(x))               # prints 3
```

List

```
x = ['pig', 'cow', 'horse']
print (len(x))               # prints 3
```

- **minimum**
  - Find the minimum item in a sequence lexicographically
  - alpha or numeric types, but cannot mix types

String

```
x = 'bug'
print (min(x))                             # prints 'b'
```

List

```
x = ['pig', 'cow', 'horse']
print (min(x))                             # prints 'cow'
```

- **maximum**
  - Find the maximum item in a sequence
  - alpha or numeric types, but cannot mix types

String

```
x = 'bug'
print (max(x))                # prints 'u'
```

List

```
x = ['pig', 'cow', 'horse']
print (max(x))                # prints 'pig'
```

- **sum**
  - Find the sum of items in a sequence
  - entire sequence must be numeric type

String -> Error

```
x = [5, 7, 'bug‘]
print (sum(x))                # error!
```

List

```
x = [2, 5, 8, 12]
print (sum(x))                # prints 27
print (sum(x[-2:]))           # prints 20
```

- **sorting**
  - Returns a new list of items in **sorted** order
  - Does not change the original list

String

```
x = 'bug'
print (sorted(x))          # prints ['b', 'g', 'u']
```

List

```
x = ['pig', 'cow', 'horse']
print (sorted(x))        # prints ['cow', 'horse', 'pig']
```

- **count (item)**
  - Returns count of an item

String

```
x = 'hippo'
print (x.count('p'))               # prints 2
```

List

```
x = ['pig', 'cow', 'horse', 'cow']
print (x.count('cow'))             # prints 2
```

- **index (item)**
  - Returns the index of the first occurrence of an item

String

```
x = 'hippo'
print (x.index('p'))               # prints 2
```

List

```
x = ['pig', 'cow', 'horse', 'cow']
print (x.index('cow'))             # prints 1
```

- **unpacking**
  - Unpack the n items of a sequence into n variables

```
x = ['pig', 'cow', 'horse']
a, b, c = x                  # now a is 'pig'
                             # b is 'cow',
                             # c is 'horse'
```

Note:

The number of variables must exactly match the length of the list.

LISTS

All operations from Sequences, plus:

• constructors	
• del list1[2]	delete item from list1
• list1.append(item)	appends an item to list1
• list1.extend(sequence1)	appends a sequence to list1
• list1.insert(index, item)	inserts item at index
• list1.pop()	pops last item
• list1.remove(item)	removes first instance of item
• list1.reverse()	reverses list order
• list1.sort()	sorts list in place
• list1.clear()	empties list

LISTS

- **constructors – creating a new list**

```
x = list((1, 2, 3))            # note double parens
x = ['a', 25, 'dog', 8.43]
x = list(tuple1)
```

List Comprehension:

```
x = [m for m in range(8)]
```

resulting list: `[0, 1, 2, 3, 4, 5, 6, 7]`

```
x = [z**2 for z in range(10) if z>4]
```

resulting list: `[25, 36, 49, 64, 81]`

LISTS

- **delete**
  - Delete a list or an item from a list

```
x = [5, 3, 8, 6]
del(x[1])              # [5, 8, 6]

del(x)                 # deletes list x
```

LISTS

- **append**
  - Append an item to a list

```
x = [5, 3, 8, 6]
x.append(7)            # [5, 3, 8, 6, 7]
```

LISTS

- **extend**
  - Append an sequence to a list

```
x = [5, 3, 8, 6]
y = [12, 13]
x.extend(y)            # [5, 3, 8, 6, 7, 12, 13]
```

LISTS

- **insert**
  - Insert an item at given index `x.insert(index, item)`

```
x = [5, 3, 8, 6]
x.insert(1, 7)            # [5, 7, 3, 8, 6]

x.insert(1,['a','m'])  # [5, ['a', 'm'], 7, 3, 8, 6]
```

LISTS

- **pop**
  - Pops last item off the list, and returns item

```
x = [5, 3, 8, 6]
x.pop()          # [5, 3, 8]
                 # and returns the 6

print(x.pop())   # prints 8
                 # x is now [5, 3]
```

LISTS

- **remove**
  - Remove first instance of an item

```
x = [5, 3, 8, 6, 3]
x.remove(3)            # [5, 8, 6, 3]
```

LISTS

- **reverse**
  - Reverse the order of the list

```
x = [5, 3, 8, 6]
x.reverse()                 # [6, 8, 3, 5]
```

LISTS

- **sort**
  - Sort the list in place

```
x = [5, 3, 8, 6]
x.sort()                      # [3, 5, 6, 8]
```

Note:

sorted(x) returns a *new* sorted list without changing the original list x.

x.sort() puts the items of x in sorted order (sorts in place).

LISTS

- **clear**
  - delete all items from the list

```
x = [5, 3, 8, 6]
x.clear()              # []
```

TUPLES

- Support all operations for Sequences
- Immutable, but member objects may be mutable
- If the contents of a list shouldn’t change, use a tuple to prevent items from accidently being added, changed or deleted
- Tuples are more efficient than lists due to Python’s implementation

TUPLES

- **constructors – creating a new tuple**

```
x = ()            # no-item tuple
x = (1,2,3)
x = 1, 2, 3       # parenthesis are optional
x = 2,            # single-item tuple
x = tuple(list1)  # tuple from list
```

TUPLES

- **immutable**
  - But member objects may be mutable

```
x = (1, 2, 3)
del(x[1])              # error!
x[1] = 8               # error!

x = ([1,2], 3)         # 2-item tuple: list and int
del(x[0][1])           # ([1], 3)
```

SETS

- **constructors – creating a new set**

```
x = {3,5,3,5}               # {5, 3}
x = set()                   # empty set
x = set(list1)              # new set from list
                            # strips duplicates
```

Set Comprehension:

```
x = {3*x for x in range(10) if x>5}
    resulting set: {18, 21, 24, 27} but in random order
```

SETS

- **basic set operations**

Description	Code
Add item to set x	`x.add(item)`
Remove item from set x	`x.remove(item)`
Get length of set x	`len(x)`

Check membership in x	`item in x`
	`item not in x`

Pop random item from set x	`x.pop()`
Delete all items from set x	`x.clear()`

SETS

- **standard mathematical set operations**

Set Function	Description	Code
Intersection	AND	`set1 & set2`
Union	OR	`set1 | set2`
Symmetric Difference	XOR	`set1 ^ set2`
Difference	In set1 but not in set2	`set1 – set2`
Subset	set2 contains set1	`set1 <= set2`
Superset	set1 contains set2	`set1 >= set2`

DICTIONARIES

- **constructors – creating a new dict**

```
x = {'pork':25.3, 'beef':33.8, 'chicken':22.7}
x = dict([('pork', 25.3),('beef', 33.8),('chicken', 22.7)])
x = dict(pork=25.3, beef=33.8, chicken=22.7)
```

DICTIONARIES

- **basic dict operations**

Description	Code
Add or change item in dict x	`x['beef'] = 25.2`
Remove item from dict x	`del x['beef']`
Get length of dict x	`len(x)`

Check membership in x
(only looks in keys, not values)
	`item in x`
	`item not in x`

Delete all items from dict x	`x.clear()`
Delete dict x	`del x`

DICTIONARIES

- **accessing keys and values in a dict**

```
x.keys()      # returns list of keys in x
x.values()    # returns list of values in x
x.items()     # returns list of key-value tuple pairs in x

item in x.values()  # tests membership in x: returns boolean
```

- **iterating a dict**

```
for key in x:                     # iterate keys
    print(key, x[key])            # print all key/value pairs

for k, v in x.items():            # iterate key/value pairs
    print(k, v)                   # print all key/value pairs
```

Note:

Entries in a dict are in random order.

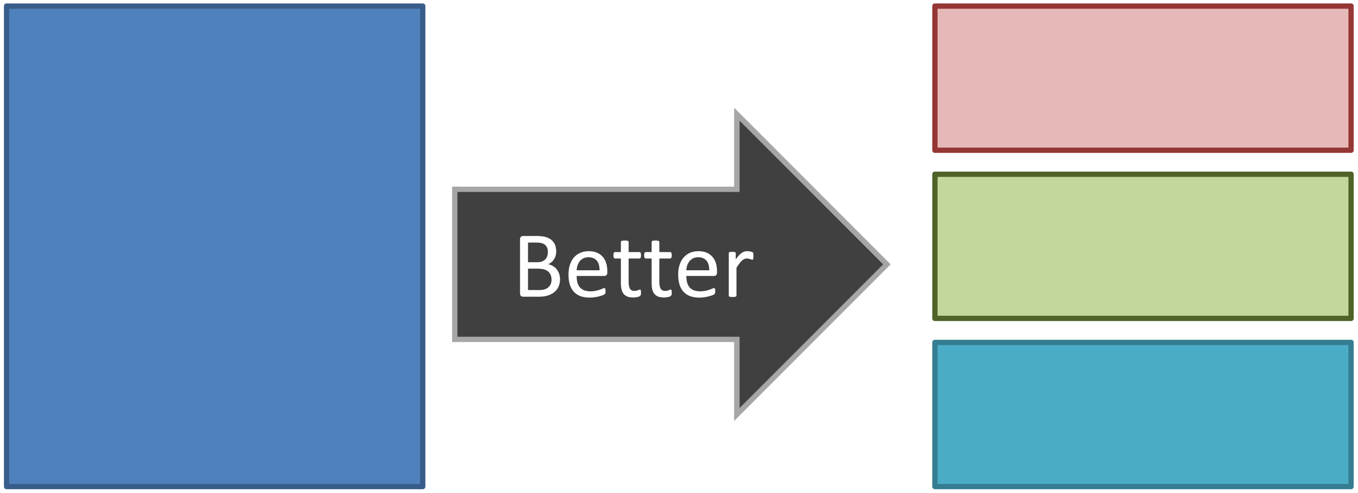
A ***function*** is a block of statements that together perform an operation.

Any operation that is used often in a program can be split into its own function.

Function benefits:

Modularizes code

Easier to **debug, re-use, maintain**

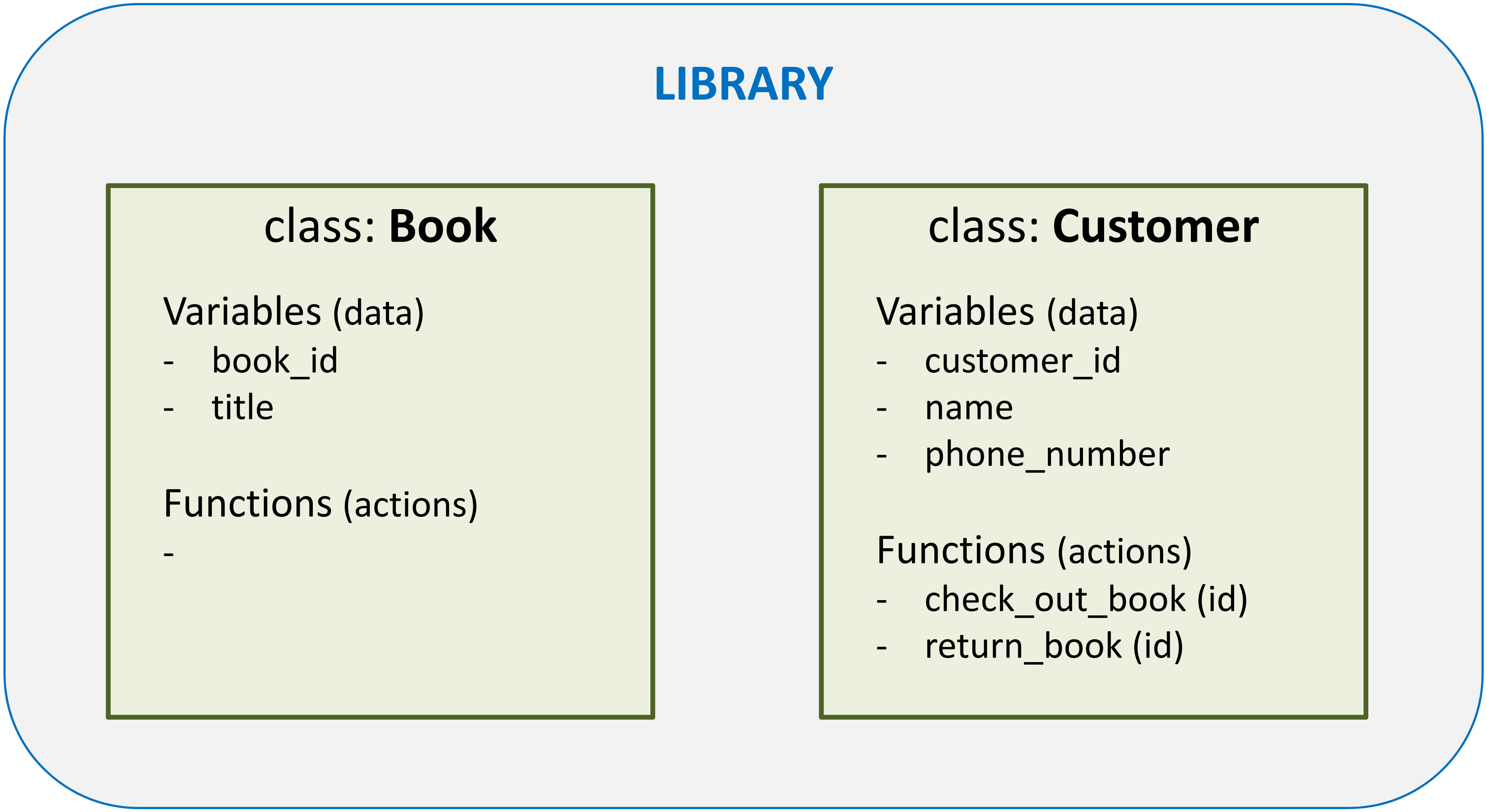


Contiguous Code

Modular Code

Classes and Objects

- Often programs must model the real world.
  - A school has courses, instructors and students.
  - A library has books and users.
- In programming this is done using *Classes*.
  - A *class* combines related **variables** and **functions** for a real-world object into one code block.
  - A class is the block of code, which is the blueprint or recipe for making an object. An object is an instance of a class (Instructor is a *class*, Professor Lee is an *object* of type Instructor)



LIBRARY

class: **Book**

Variables (data)

- book_id
- title

Functions (actions)

-

class: **Customer**

Variables (data)

- customer_id
- name
- phone_number

Functions (actions)

- check_out_book (id)
- return_book (id)