Imperative programming with Python

Class #4

Facundo Carreiro

ILLC, University of Amsterdam

January 2015

Data structures

- A *data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Data structures

- A *data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- We have already stumbled upon one of them

```
>>> L = [2,3,5,7]
>>> type(L)
<type 'list'>
```

  The *List* data type!

- The values of the list type are sequences of elements $a_1, \ldots, a_n$,
- Where each a_i is a value of any type.

Data structures: Lists

- The easiest way to create a list is using the square brackets

```
L = []
```

is the empty list, and

```
L = [2, 'hello', [4, True], abs(-1)]
```

is an example of a nested list.

Data structures: Lists

- The easiest way to create a list is using the square brackets

```
L = []
```

  is the empty list, and

```
L = [2, 'hello', [4, True], abs(-1)]
```

  is an example of a nested list.

- You can index them as you did with strings

```
>>> L[1]
'hello'
>>> L[2][1]
True
```

Data structures: Lists

- The easiest way to create a list is using the square brackets

```
L = []
```

  is the empty list, and

```
L = [2, 'hello', [4, True], abs(-1)]
```

  is an example of a nested list.

- You can index them as you did with strings

```
>>> L[1]
'hello'
>>> L[2][1]
True
```

- The `len(·)` function, as usual, returns the length of the list

```
>>> len(L)
4
```

Data structures: Lists

- Lists *are* mutable

```
>>> L[0] = 5*5
>>> L
[25, 'hello', [4, True], 1]
```

Data structures: Lists

- Lists *are* mutable

```
>>> L[0] = 5*5
>>> L
[25, 'hello', [4, True], 1]
```

- You can use `+` to concatenate lists

```
>>> [1,2] + [3,4] + [5]
[1, 2, 3, 4, 5]
```

Data structures: Lists

- Lists *are* mutable

```
>>> L[0] = 5*5
>>> L
[25, 'hello', [4, True], 1]
```

- You can use `+` to concatenate lists

```
>>> [1,2] + [3,4] + [5]
[1, 2, 3, 4, 5]
```

- You can use `+` and the `append` and `insert` methods to add elements to a list (among others)

```
>>> L + [3]
[25, 'hello', [4, True], 1, 3]
>>> L.append(6)
>>> L
[25, 'hello', [4, True], 1, 6]
```

Data structures: Lists

- Lists *are* mutable

```
>>> L[0] = 5*5
>>> L
[25, 'hello', [4, True], 1]
```

- You can use `+` to concatenate lists

```
>>> [1,2] + [3,4] + [5]
[1, 2, 3, 4, 5]
```

- You can use `+` and the `append` and `insert` methods to add elements to a list (among others)

```
>>> L + [3]
[25, 'hello', [4, True], 1, 3]
>>> L.append(6)
>>> L
[25, 'hello', [4, True], 1, 6]
```

- Question: where did the '3' go?

Data structures: Lists

- There are several ways to delete an item from a list

Data structures: Lists

- There are several ways to delete an item from a list
- If you know the index you can use `del`

```
>>> M = ['a','f','z']
>>> del M[0]
>>> M
['f', 'z']
```

or the `pop(·)` method

```
>>> M.pop(1)
'z'
>>> M
['f']
```

Data structures: Lists

- There are several ways to delete an item from a list
- If you know the index you can use `del`

```
>>> M = ['a','f','z']
>>> del M[0]
>>> M
['f', 'z']
```

or the `pop(·)` method

```
>>> M.pop(1)
'z'
>>> M
['f']
```

- If you know the element but not the index you can use the `remove(·)` method to remove the first occurrence

```
>>> M = ['a','b','b','c']
>>> M.remove('b')
>>> M
['a', 'b', 'c']
```

Data structures: Lists

- Lists can be iterated, it is one of the most common operations

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> acumm = 0
>>> for i in range(5):
...     acumm += i
>>> acumm
10
```

Data structures: Lists

- Lists can be iterated, it is one of the most common operations

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> acumm = 0
>>> for i in range(5):
...     acumm += i
>>> acumm
10
```

- The slice (`[n:m]`) operator also works with them

```
>>> L[:2]
[25, 'hello']
```

- Suggested HW: check the Python documentation for Lists.

Data structures: Lists and strings

- Strings are sequences of characters
- But that is not the same as a list of characters

```
>>> s = 'hello'
>>> l = ['h','e','l','l','o']
>>> type(s)
<type 'str'>
>>> type(l)
<type 'list'>
>>> print s, l
hello ['h', 'e', 'l', 'l', 'o']
```

Data structures: Lists and strings

- Strings are sequences of characters
- But that is not the same as a list of characters

```
>>> s = 'hello'
>>> l = ['h','e','l','l','o']
>>> type(s)
<type 'str'>
>>> type(l)
<type 'list'>
>>> print s, l
hello ['h', 'e', 'l', 'l', 'o']
```

- The `list(·)` function converts strings to lists

```
>>> list(s)
['h', 'e', 'l', 'l', 'o']
```

Data structures: Lists and strings

- A much more interesting effect can be achieved using the `split` string method

```
>>> 'what␣a␣wonderful␣world'.split()
['what', 'a', 'wonderful', 'world']
```

Keep this one in mind, it's very useful.
Suggested HW: execute `help('any string'.split)`

Data structures: Lists and strings

- A much more interesting effect can be achieved using the `split` string method

```
>>> 'what␣a␣wonderful␣world'.split()
['what', 'a', 'wonderful', 'world']
```

Keep this one in mind, it's very useful.
Suggested HW: execute `help('any string'.split)`

- To do the inverse, you use the `join` function of the `string` module

```
>>> import string
>>> string.join(['put', 'us', 'toghether'])
'put␣us␣toghether'
>>> string.join(['first','second','third'],',␣')
'first,␣second,␣third'
```

The Object Model

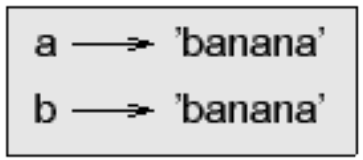
- We said that variables referred to values, but actually that is not true.
- Variables refer to *objects*.
- Objects are abstractions for data, they have
  1. A type
  2. An identity (can be though of as: "the place in the memory")
  3. A value

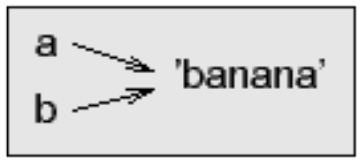
The Object Model

- We said that variables referred to values, but actually that is not true.
- Variables refer to *objects*.
- Objects are abstractions for data, they have
  1. A type
  2. An identity (can be though of as: "the place in the memory")
  3. A value

Let's analyze how the following piece of code acts

```
a = 'banana'
b = 'banana'
```



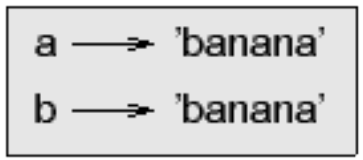


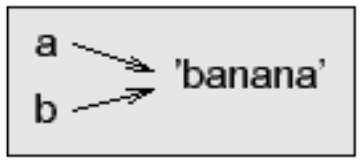
The Object Model

- We said that variables referred to values, but actually that is not true.
- Variables refer to *objects*.
- Objects are abstractions for data, they have
  1. A type
  2. An identity (can be though of as: "the place in the memory")
  3. A value

Let's analyze how the following piece of code acts

```
a = 'banana'
b = 'banana'
```





The `is` operator compares *objects* and tells us we are in the second case.

```
>>> a is b
True
```

```
>>> a == b
True
```

The Object Model

Let's see what happens with Lists

```
a = [1, 2, 3]
b = [1, 2, 3]
```

The Object Model

Let's see what happens with Lists

```
a = [1, 2, 3]
b = [1, 2, 3]
```

We use the `is` and `==` operators to test it

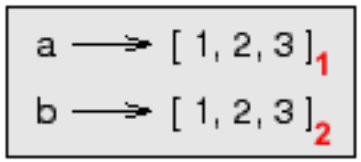
```
>>> a is b
False
```

```
>>> a == b
True
```

The Object Model

Let's see what happens with Lists

```
a = [1, 2, 3]
b = [1, 2, 3]
```



We use the `is` and `==` operators to test it

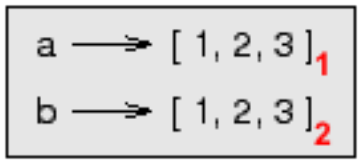
```
>>> a is b
False
```

```
>>> a == b
True
```

The Object Model

Let's see what happens with Lists

```
a = [1, 2, 3]
b = [1, 2, 3]
```



We use the `is` and `==` operators to test it

```
>>> a is b
False
```

```
>>> a == b
True
```

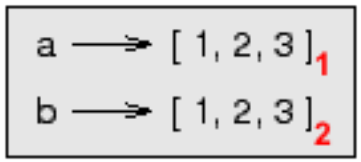
What happens in the following case?

```
a = [1, 2, 3]
b = a
```

The Object Model

Let's see what happens with Lists

```
a = [1, 2, 3]
b = [1, 2, 3]
```



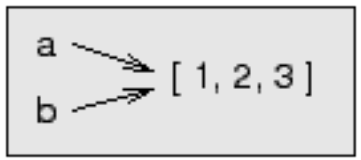
We use the `is` and `==` operators to test it

```
>>> a is b
False
```

```
>>> a == b
True
```

What happens in the following case?

```
a = [1, 2, 3]
b = a
```



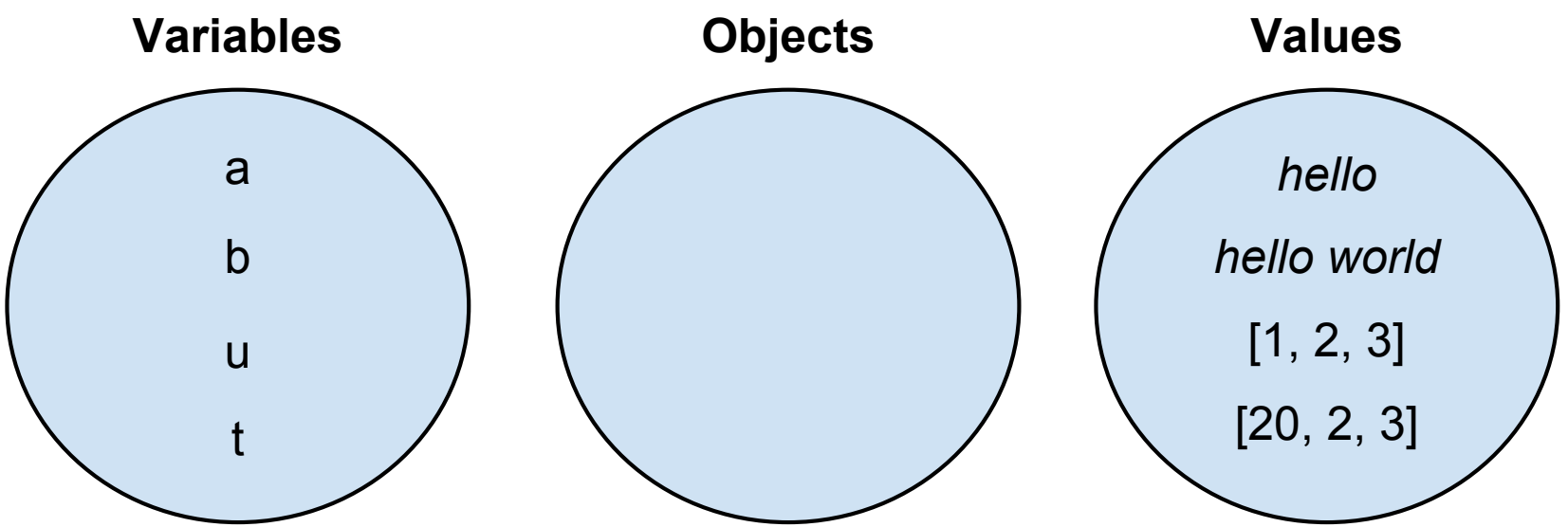

```
>>> a is b
True
```

```
>>> a == b
True
```

`a` and `b` refer to the same object. They are called *aliases*.

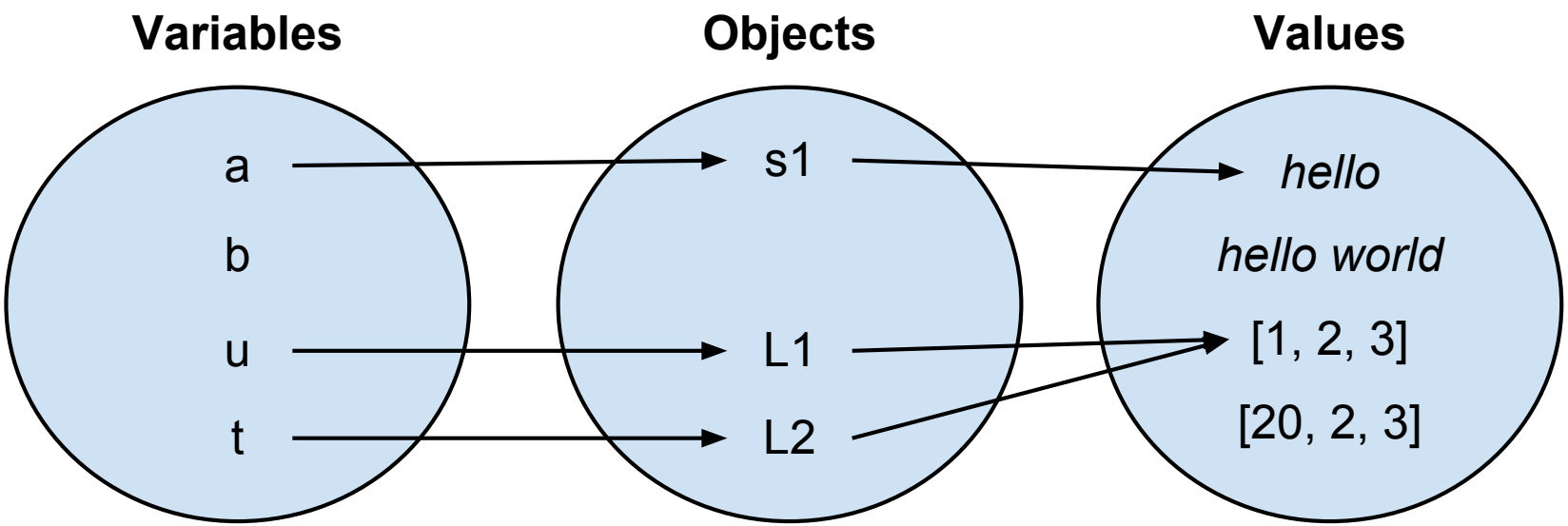
The Object Model: aliasing

- ```
  a = 'hello'; t = [1,2,3]; u = [1,2,3]
  u = t
  t[0] = 20
  b = a
  a = a + ' world'
  ```


  ```

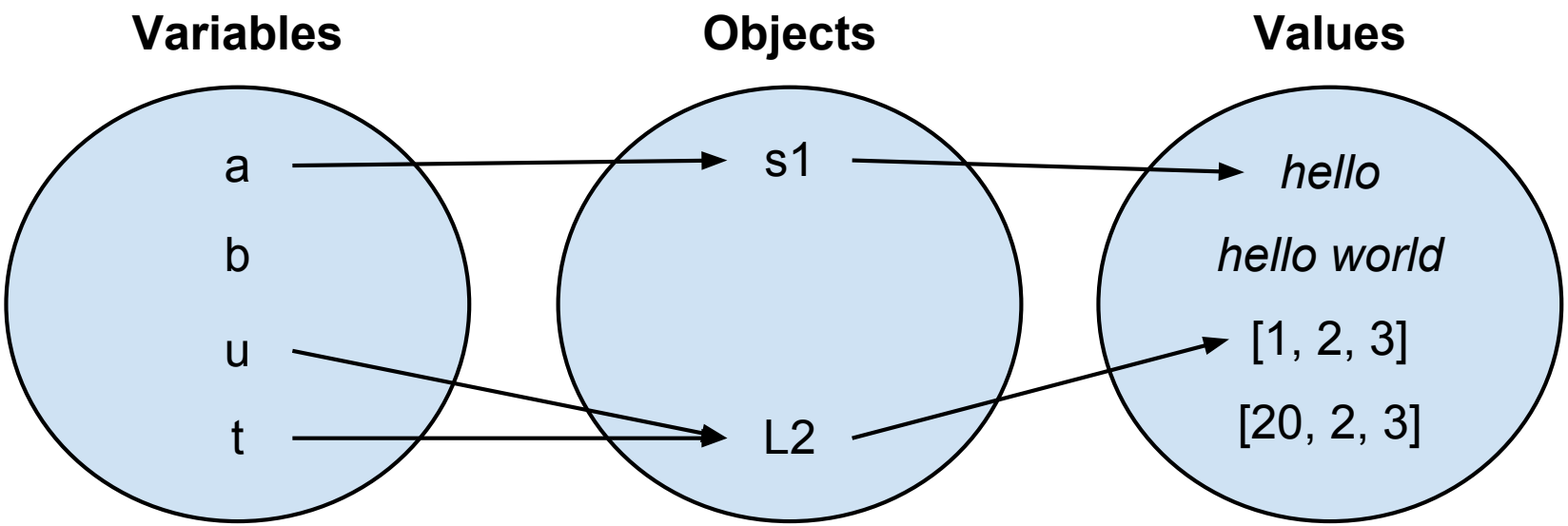
The Object Model: aliasing

```
a = 'hello'; t = [1,2,3]; u = [1,2,3]
u = t
t[0] = 20
b = a
a = a + ' world'
```



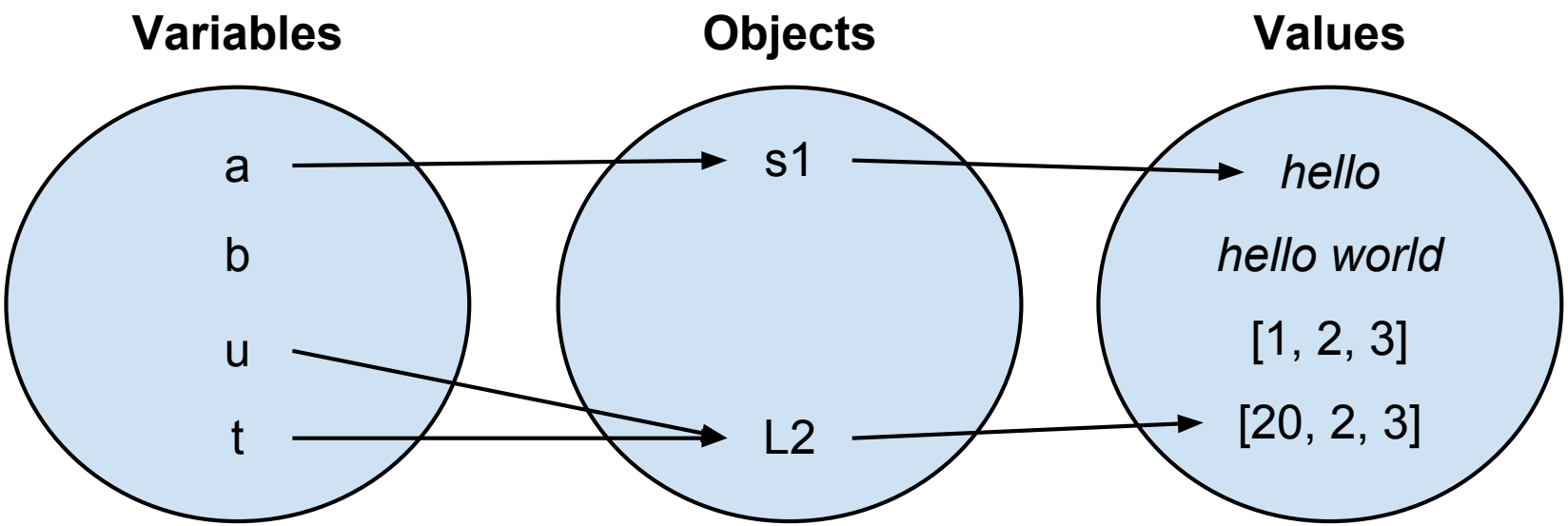
The Object Model: aliasing

```
a = 'hello'; t = [1,2,3]; u = [1,2,3]
u = t
t[0] = 20
b = a
a = a + ' world'
```



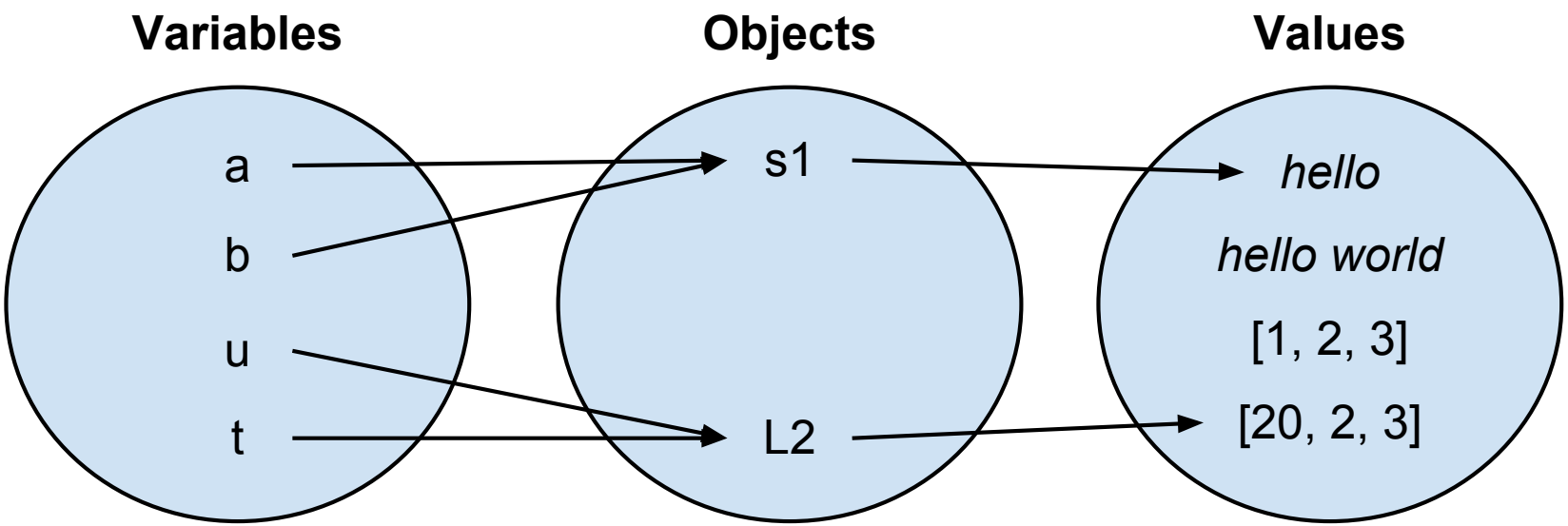
The Object Model: aliasing

```
a = 'hello'; t = [1,2,3]; u = [1,2,3]
u = t
t[0] = 20
b = a
a = a + ' world'
```



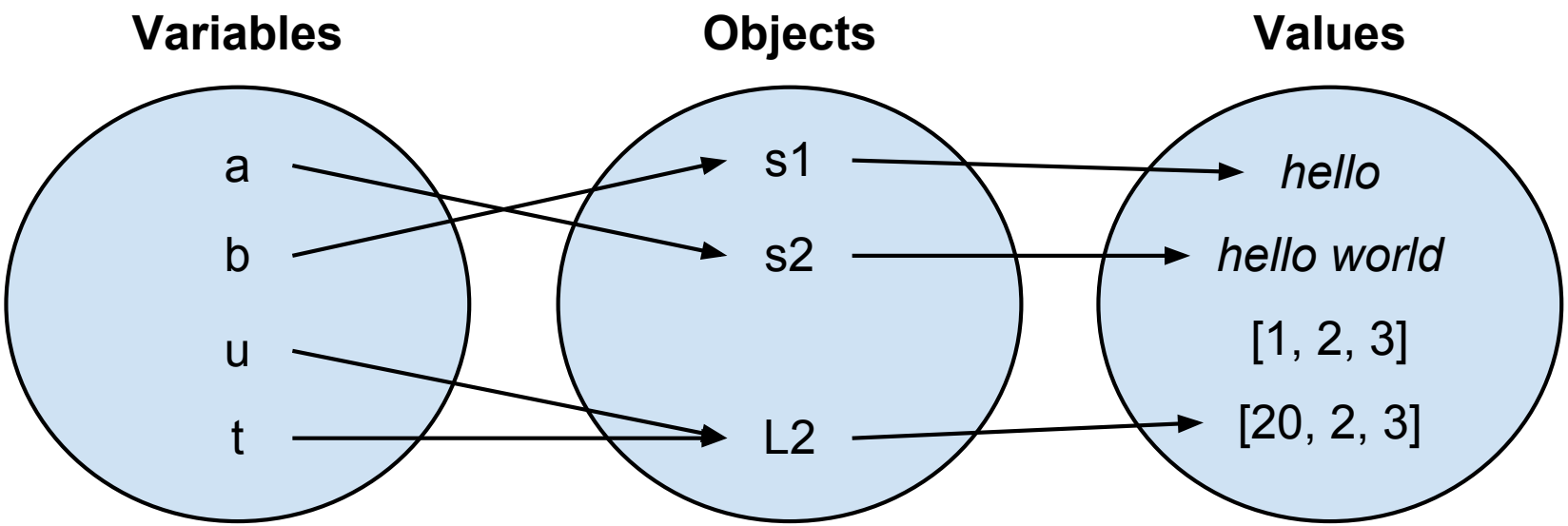
The Object Model: aliasing

```
a = 'hello'; t = [1,2,3]; u = [1,2,3]
u = t
t[0] = 20
b = a
a = a + ' world'
```



The Object Model: aliasing

```
a = 'hello'; t = [1,2,3]; u = [1,2,3]
u = t
t[0] = 20
b = a
a = a + ' world'
```



Data Structures: tuples

- *Tuples* are fixed length, *immutable* sequences of items.
- You use commas and (optionally) parentheses to create them

```
>>> t = (55, 'text', 8)
>>> u = (4,)
>>> v = (4)
```

```
>>> type(t)
<type 'tuple'>
>>> type(u)
<type 'tuple'>
>>> type(v)
<type 'int'>
```

Observe that to get a 1-tuple we need to add an extra comma.

- They can be indexed, iterated and sliced just as lists and strings.

Data Structures: tuples

- Accessing each item of a tuple could be annoying

```
t = [(1,2,3), ('a','b','c')]
for e in t:
  x = e[0]
  y = e[1]
  z = e[2]
  print x + y + z
```

Data Structures: tuples

- Accessing each item of a tuple could be annoying

```
t = [(1,2,3), ('a','b','c')]
for e in t:
  x = e[0]
  y = e[1]
  z = e[2]
  print x + y + z
```

- Luckily, tuples can be handled in a very handy way

```
t = [(1,2,3), ('a','b','c')]
for (x,y,z) in t:
  print x + y + z
```

```
addr = 'monty@python.org'
(uname, domain) = addr.split('@')
```

Data Structures: tuples

- Accessing each item of a tuple could be annoying

```
t = [(1,2,3), ('a','b','c')]
for e in t:
  x = e[0]
  y = e[1]
  z = e[2]
  print x + y + z
```

- Luckily, tuples can be handled in a very handy way

```
t = [(1,2,3), ('a','b','c')]
for (x,y,z) in t:
  print x + y + z
```

```
addr = 'monty@python.org'
(uname, domain) = addr.split('@')
```

- Side note: Functional languages usually have an extended version of this phenomenon called *pattern matching*.

Data Structures: List Comprehensions

- Python has an awesome way of constructing lists called *list comprehension*. They mimic mathematical definitions such as

$$\{f(x) \mid x \in C \wedge \text{condition\_holds}(x)\}$$

Data Structures: List Comprehensions

- Python has an awesome way of constructing lists called *list comprehension*. They mimic mathematical definitions such as

$$\{f(x) \mid x \in C \wedge \text{condition\_holds}(x)\}$$

- Some examples

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> words = ['dog', 'cat', 'yellow']
>>> [(w, len(w)) for w in words if 'a' not in w]
[('dog', 3), ('yellow', 6)]
```

- Suggested HW: Check the reference for more involved examples.

Data Structures: Dictionaries

- A *dictionary* is a group of ($key \mapsto value$) assignments.
- The empty dictionary may be created with `{}` or `dict(·)`.

```
>>> d1 = {}
>>> d2 = dict()
```

Data Structures: Dictionaries

- A *dictionary* is a group of ($key \mapsto value$) assignments.
- The empty dictionary may be created with `{}` or `dict(·)`.

```
>>> d1 = {}
>>> d2 = dict()
```

- You can create a dictionary with some predefined assignments.

```
d = {1:'mom', 2:'god',
(25,17):"[...] And you will know that my name is the Lord \
when I lay my vengeance upon thee."}
```

$$\begin{aligned} 1 &\mapsto \textit{mom} \\ 2 &\mapsto \textit{god} \\ (25, 17) &\mapsto \textit{[...] And you will know that my name...} \end{aligned}$$

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

- You can 'index' the dictionary using it's keys

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

- You can 'index' the dictionary using it's keys

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

- You can 'index' the dictionary using it's keys

```
>>> d[1]
'mom'
```

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

- You can 'index' the dictionary using it's keys

```
>>> d[1]
'mom'
```

```
>>> d[0]
KeyError: 0
```

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

- You can 'index' the dictionary using it's keys

```
>>> d[1]
'mom'
```

```
>>> d[0]
KeyError: 0
```

- You can also create or update a key-value pair using `[·]`.

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

- You can 'index' the dictionary using it's keys

```
>>> d[1]
'mom'
```

```
>>> d[0]
KeyError: 0
```

- You can also create or update a key-value pair using `[·]`.

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

- You can 'index' the dictionary using it's keys

```
>>> d[1]
'mom'
```

```
>>> d[0]
KeyError: 0
```

- You can also create or update a key-value pair using `[·]`.

```
>>> d[1] = True
```

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

- You can 'index' the dictionary using it's keys

```
>>> d[1]
'mom'
```

```
>>> d[0]
KeyError: 0
```

- You can also create or update a key-value pair using `[·]`.

```
>>> d[1] = True
```

```
>>> d[0] = 'mom'
```

Data Structures: Dictionaries

- The `has_key(·)` method tells you if the key is defined

```
>>> d.has_key(1)
True
```

- You can 'index' the dictionary using it's keys

```
>>> d[1]
'mom'
```

```
>>> d[0]
KeyError: 0
```

- You can also create or update a key-value pair using `[·]`.

```
>>> d[1] = True
```

```
>>> d[0] = 'mom'
```

- Deletion is achieved through the `del` statement as in lists.

Data Structures: Dictionaries

- The `keys`, `values` and `iteritems` methods let you iterate over the dictionary

```
>>> knights = {'gallahad': 'the␣pure', 'robin': 'the␣brave'}
>>> print knights.keys()
['gallahad', 'robin']
```

```
>>> knights.values()
['the␣pure', 'the␣brave']
```

Again, we can use pattern matching with tuples

```
>>> for (k, v) in knights.iteritems():
...     print k + ',␣so␣called␣' + v
...
gallahad, so called the pure
robin, so called the brave
```

References

- Chapters 10–12 of the book
  http://greenteapress.com/thinkpython/thinkpython.html
- List Methods
  http://docs.python.org/tutorial/datastructures.html#more-on-lists
- Python Data Model
  http://docs.python.org/reference/datamodel.html
- List Comprehensions
  http://docs.python.org/tutorial/datastructures.html#list-comprehensions
- Dictionaries
  http://docs.python.org/tutorial/datastructures.html#dictionaries