COMP110: Principles of Computing

8: Primitive Data Types

Learning outcomes

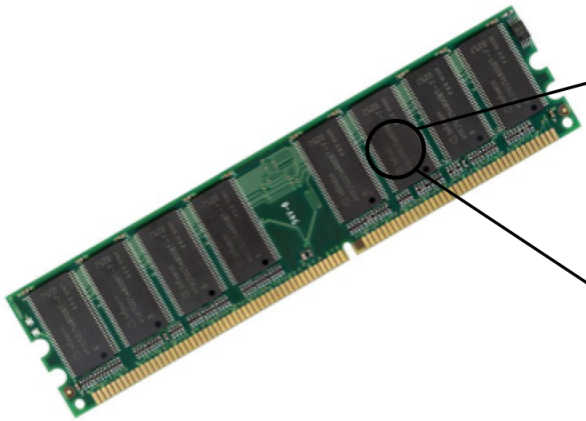
- **Explain** the representation of common "plain old data" types in memory
- **Distinguish** pass-by-reference and pass-by-value
- **Distinguish** allocation of memory on the stack and on the heap

FALMOUTH
UNIVERSITY

Data representation

Memory

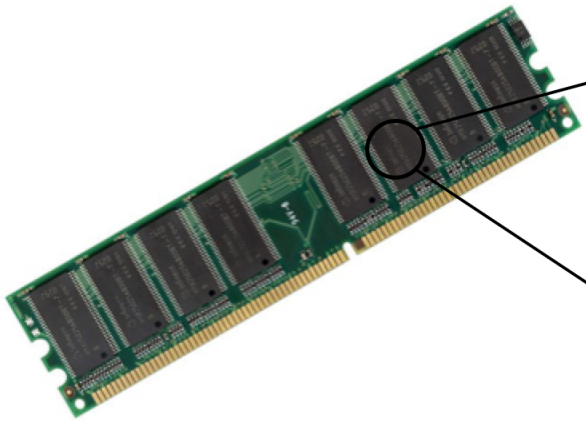
Memory



Address	Data
0	01001000
1	01100101
2	01101100
3	01101100
4	01101111
5	00100001
...	...

- Memory works like a set of **boxes**

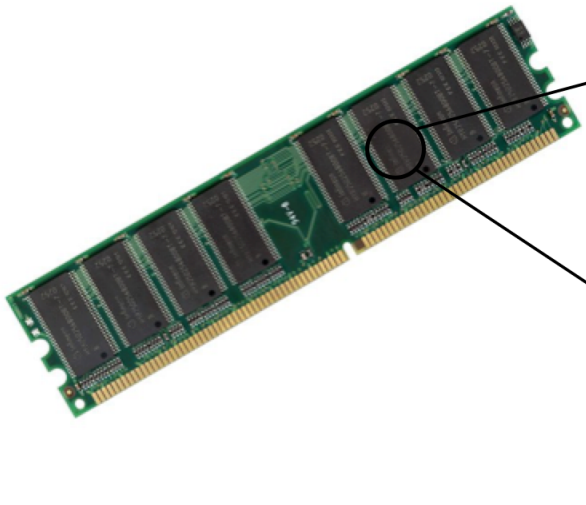
Memory



Address	Data
0	01001000
1	01100101
2	01101100
3	01101100
4	01101111
5	00100001
...	...

- Memory works like a set of **boxes**
- Each box has a number, its **address**

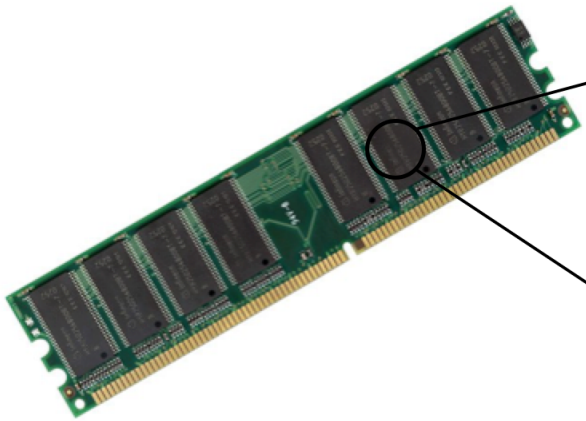
Memory



Address	Data
0	01001000
1	01100101
2	01101100
3	01101100
4	01101111
5	00100001
...	...

- Memory works like a set of **boxes**
- Each box has a number, its **address**
- Each box contains a **byte** (8 bits)

Memory



Address	Data
0	01001000
1	01100101
2	01101100
3	01101100
4	01101111
5	00100001
...	...

- Memory works like a set of **boxes**
- Each box has a number, its **address**
- Each box contains a **byte** (8 bits)
- All data is stored as **sequences of bytes**

Integers

FALMOUTH
UNIVERSITY

Pass by reference

References

References

- Our picture of a variable: a labelled box containing a value

References

- Our picture of a variable: a labelled box containing a value
- For "plain old data" (e.g. numbers), this is accurate

References

- Our picture of a variable: a labelled box containing a value
- For "plain old data" (e.g. numbers), this is accurate
- For **objects** (i.e. instances of classes), variables actually hold **references** (a.k.a. **pointers**)

References

- Our picture of a variable: a labelled box containing a value
- For "plain old data" (e.g. numbers), this is accurate
- For **objects** (i.e. instances of classes), variables actually hold **references** (a.k.a. **pointers**)
- It is possible (indeed common) to have **multiple references** to the same underlying object

The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value
x	
y	
z	

The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value
x	a: 30 b: 40
y	
z	

The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value	
x	a	30
	b	40
y	a	50
	b	60
z		

The wrong picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value	
x	a	30
	b	40
y	a	50
	b	60
z	a	50
	b	60

The right picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

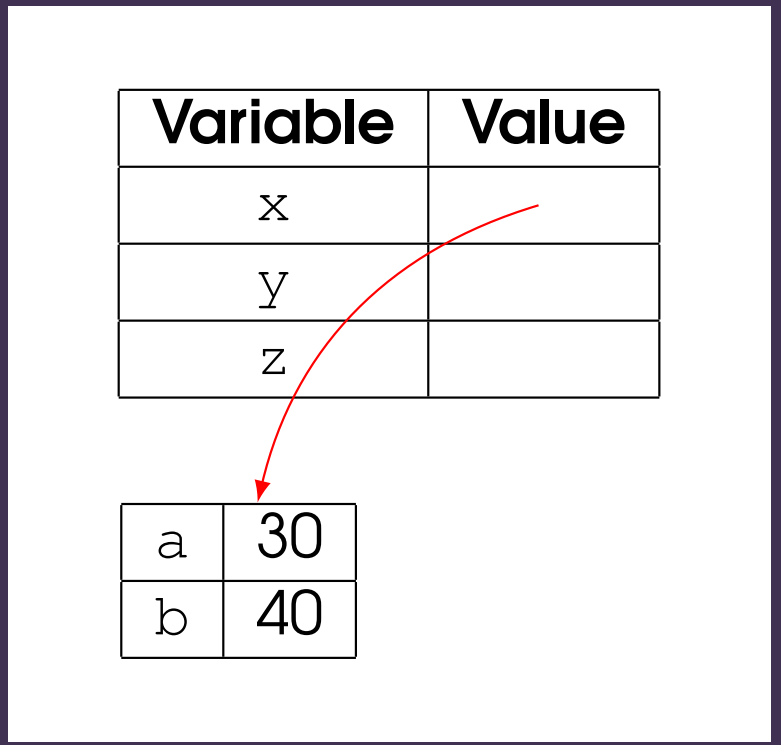
The right picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```

Variable	Value
x	
y	
z	

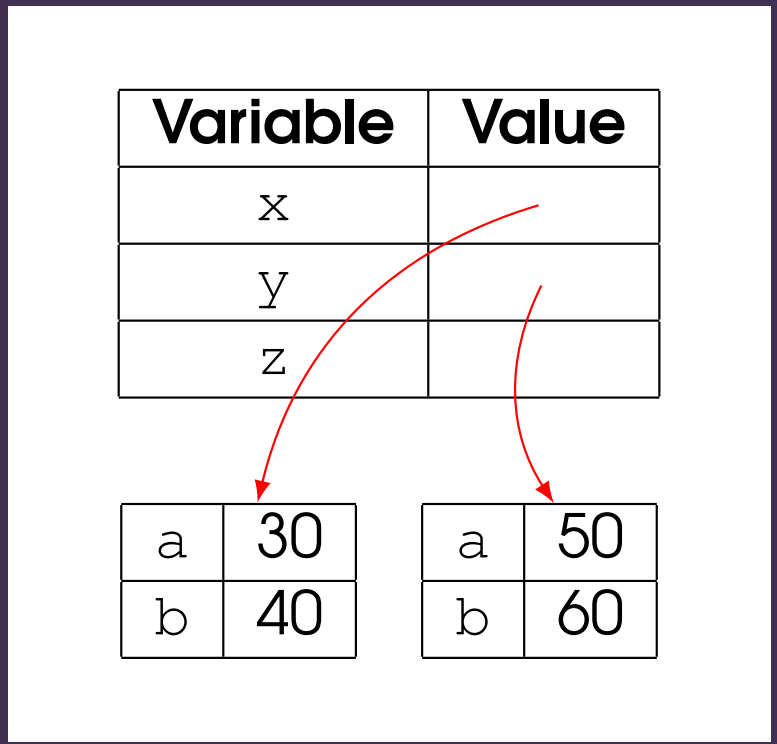
The right picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```



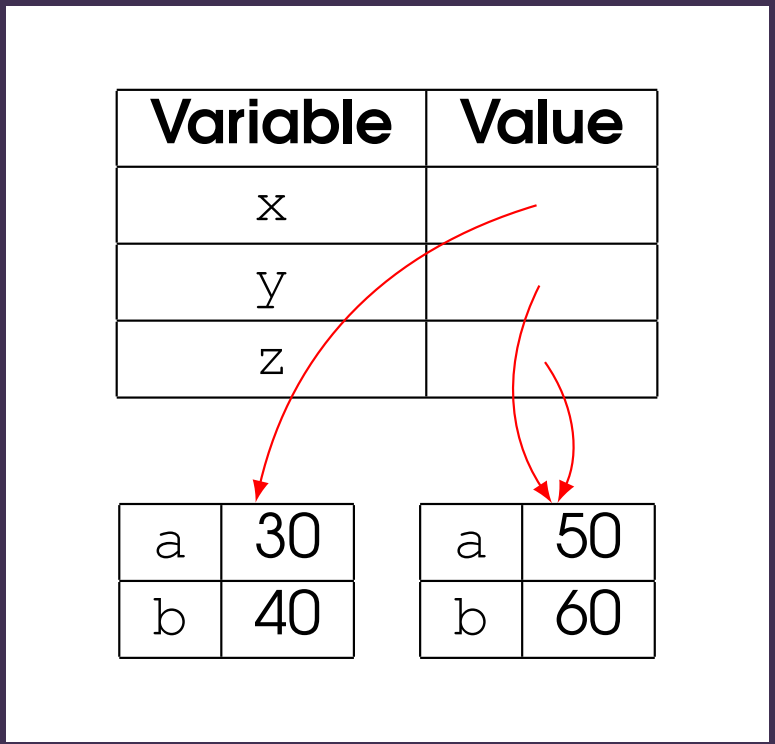
The right picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```



The right picture

```
class Thing:
    def __init__(self,
                 a, b):
        self.a = a
        self.b = b

x = Thing(30, 40)
y = Thing(50, 60)
z = y
```



Values and references

Socrative room code: `FALCOMPED`

```
a = 10
b = a
a = 20
print "a:", a
print "b:", b
```

Values and references

Socrative room code: `FALCOMPED`

```
a = 10
b = a
a = 20
print "a:", a
print "b:", b
```

Variable	Value
a	
b	

Values and references

Socrative room code: FALCOMPED

```
class X:
    def __init__(self, value):
        self.value = value

a = X(10)
b = a
a.value = 20
print "a:", a.value
print "b:", b.value
```

Values and references

Socrative room code: FALCOMPED

```
class X:
    def __init__(self, value):
        self.value = value

a = X(10)
b = a
a.value = 20
print "a:", a.value
print "b:", b.value
```

Variable	Value
a	
b	

Values and references

Socrative room code: FALCOMPED

```
class X:
    def __init__(self, value):
        self.value = value

a = X(10)
b = X(10)
a.value = 20
print "a:", a.value
print "b:", b.value
```

Values and references

Socrative room code: FALCOMPED

```
class X:
    def __init__(self, value):
        self.value = value

a = X(10)
b = X(10)
a.value = 20
print "a:", a.value
print "b:", b.value
```

Variable	Value
a	
b	

Pass by value

Pass by value

In **function parameters**, "plain old data" is passed by **value**

Pass by value

In **function parameters**, "plain old data" is passed by **value**

```
def double(x):
    x *= 2

a = 7
double(a)
print a
```

Pass by value

In **function parameters**, "plain old data" is passed by **value**

```
def double(x):
    x *= 2

a = 7
double(a)
print a
```

`double` does not actually do anything, as `x` is just a local copy of whatever is passed in!

Pass by reference

Pass by reference

However, instances are passed by **reference**

```
class Box:
    def __init__(self, v):
        self.value = v

def double(x):
    x.value *= 2

a = Box(7)
double(a)
print a.value
```

Pass by reference

However, instances are passed by **reference**

```
class Box:
    def __init__(self, v):
        self.value = v

def double(x):
    x.value *= 2

a = Box(7)
double(a)
print a.value
```

`double` now has an effect, as `x` gets a reference to the `Box` instance

Lists are objects too

Lists are objects too

```
a = ["Hello"]
b = a
b.append("world")
print a  # ["Hello", "world"]
```

Lists are objects too

```
a = ["Hello"]
b = a
b.append("world")
print a   # ["Hello", "world"]
```

... which means you should be careful when passing lists into functions, because the function might actually change the list!