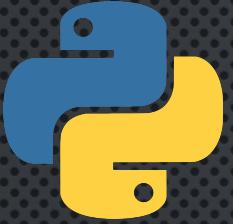


PYTHON DEEP DIVE

PART 1



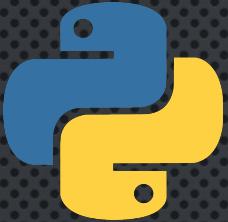
This course is about

the Python language and built-in types

the standard library

becoming an expert Python developer

idiomatic Python



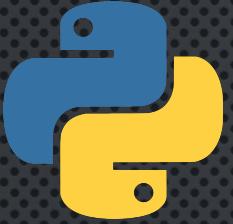
The Zen of Python

Tim Peters

`import this`

PEP 20

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than **right** now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!



Quick Refresher

Python's type hierarchy

variables

conditionals

functions

loops

break, continue and try

classes



Variables and Memory

what are variables? → symbols for memory addresses

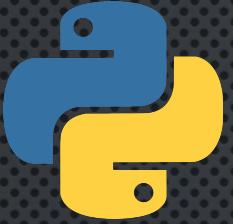
memory

Python memory management → reference counting, garbage collection

mutability → function arguments, shared references

what is equality of two objects?

Python memory optimizations → interning, peephole



Numeric Types

integers

rationals

floats

→ binary representations exactness rounding

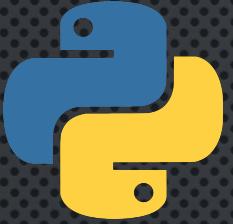
equality measures of closeness, approximate equality

Decimals

→ alternative to floats exactness precision rounding

complex numbers

→ `cmath` standard library



Numeric Types - Booleans

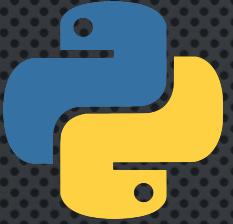
associated Truth values → every object has one

precedence and short-circuiting

Boolean operators → what they really do

using in context of associated truth values

comparison operators → identity, value equalities ordering



Functions

higher-order functions

docstrings and annotations

Lambdas

introspection

functional programming

- map, filter, zip
- reducing functions
- partial functions



Functions - Arguments

positional arguments

keyword-only arguments

default values → caveats

packing and unpacking

variable positional arguments

variable keyword-only arguments



Functions - Scopes and Closures

global and local scopes

nested scopes

closures

nested closures



Decorators

decorators

nested decorators

parameterized decorators

stacked decorators

class decorators

decorator classes

applications → memoization, single dispatch, logging, timing



Tuples as Data Structures

tuples are not just read-only lists

data structures

packing and unpacking

named tuples

augmenting named tuples



Modules, Packages and Namespaces

what are modules?

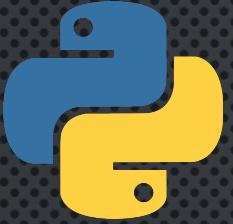
what are packages?

how do the various imports work?

how to manipulate namespaces using packages

zip archives

`__main__`



Extras

will keep growing over time

important new features of Python 3.6 and later

best practices

random collection of interesting stuff

additional resources

send me your suggestions!

QUICK REFRESHER

Python Type Hierarchy

focus on types covered in this course

Multi-Line Statements and Strings

Naming Conventions

Conditionals

Functions

Loops

While
For

Break, Continue, Try

PYTHON TYPE HIERARCHY

(SUBSET)

The following is a subset of the Python type hierarchy that we will cover in this course:

Numbers

Integral

Integers

Booleans

Non-Integral

Floats (*c doubles*)

Complex

Decimals

Fractions

Collections

Sequences

Mutable
Lists

Immutable
Tuples
Strings

Sets

Mutable
Sets

Immutable
Frozen Sets

Mappings

Dictionaries

Callables

User-Defined Functions

Generators

Classes

Instance Methods

Class Instances (`__call__()`)

Built-in Functions (e.g. `len()`, `open()`)

Built-in Methods (e.g. `my_list.append(x)`)

Singletons

None

`NotImplemented`

Ellipsis (...)

MULTI-LINE STATEMENTS AND STRINGS

Copyright © 2014

Python Program

- physical lines of code end with a physical **newline** character
- logical lines of code end with a logical **NEWLINE** token
- tokenized

physical newline vs logical newline

sometimes, physical newlines are ignored
in order to combine multiple physical lines
into a single logical line of code
terminated by a logical **NEWLINE** token

Conversion can be **implicit** or **explicit**

Implicit

Expressions in:

list literals: []

tuple literals: ()

dictionary literals: { }

set literals: { }

function arguments / parameters

supports inline comments

```
[1,  
 2,  
 3]  
[ 1, #item 1  
 2, #item 2  
 3 #item 3  
 ]
```

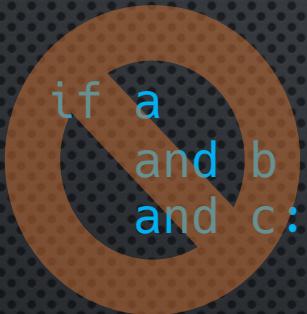
```
def my_func(a,  
           b, #comment  
           c):  
    print(a, b, c)
```

```
my_func(10, #comment  
        20, 30)
```

Explicit

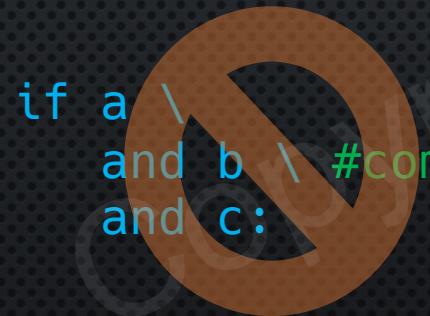
You can break up statements over multiple lines **explicitly**, by using the \ (backslash) character

Multi-line statements are not implicitly converted to a single logical line.



```
if a \
and b \
and c:
```

Comments **cannot** be part of a statement, not even a multi-line statement.



```
if a \
and b \ #comment
and c:
```

Multi-Line String Literals

Multi-line string literals can be created using triple delimiters (' single or " double)

```
'''This is  
a multi-line string'''
```

```
"""This is  
a multi-line string"""
```

Be aware that non-visible characters such as `newlines`,
`tabs`, etc. are actually part of the string – basically
anything you type.

You can use escaped characters (e.g. `\n`, `\t`), use string formatting, etc.

A multi-line string is just a regular string.

Multi-line strings are not comments, although they can be used as such,
especially with special comments called `docstrings`.

IDENTIFIER NAMES

RULES AND CONVENTIONS

Identifier names

are case-sensitive `my_var` are different identifiers

`my_Var`
`ham`
`Ham`

must follow certain rules

should follow certain conventions

Must

start with underscore (_) or letter (a-z A-Z)

followed by any number of underscores (_), letters (a-z A-Z), or digits (0-9)

var my_var index1 index_1 _var __var __lt__ are all legal names

cannot be reserved words:

None True False

and or not

if else elif

for while break continue pass

def lambda global nonlocal return yield

del in is assert class

try except finally raise

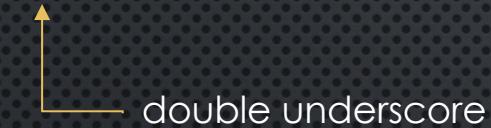
import from with as

Conventions

`_my_var`



`__my_var`



`__my_var__`



This is a convention to indicate "internal use" or "private" objects

Objects named this way will not get imported by a statement such as :
`from module import *`

Used to "mangle" class attributes – useful in inheritance chains

Used for system-defined names that have a special meaning to the interpreter.

Don't invent them, stick to the ones pre-defined by Python!

`__init__`

`x < y` \longrightarrow `x.__lt__(y)`

Other Naming Conventions

from the PEP 8 Style Guide

Packages	short, all-lowercase names. Preferably no underscores.	utilities
Modules	short, all-lowercase names. Can have underscores.	db_utils dbutils
Classes	CapWords (upper camel case) convention	BankAccount
Functions	lowercase, words separated by underscores (snake_case)	open_account
Variables	lowercase, words separated by underscores (snake_case)	account_id
Constants	all-uppercase, words separated by underscores	MIN_APY

<https://www.python.org/dev/peps/pep-0008/> ← This is a should-read!

A foolish consistency is the hobgoblin of little minds
(Emerson)

VARIABLES

memory references

what variables really are

memory management

reference counting

garbage collection

dynamic vs static typing

mutability and immutability

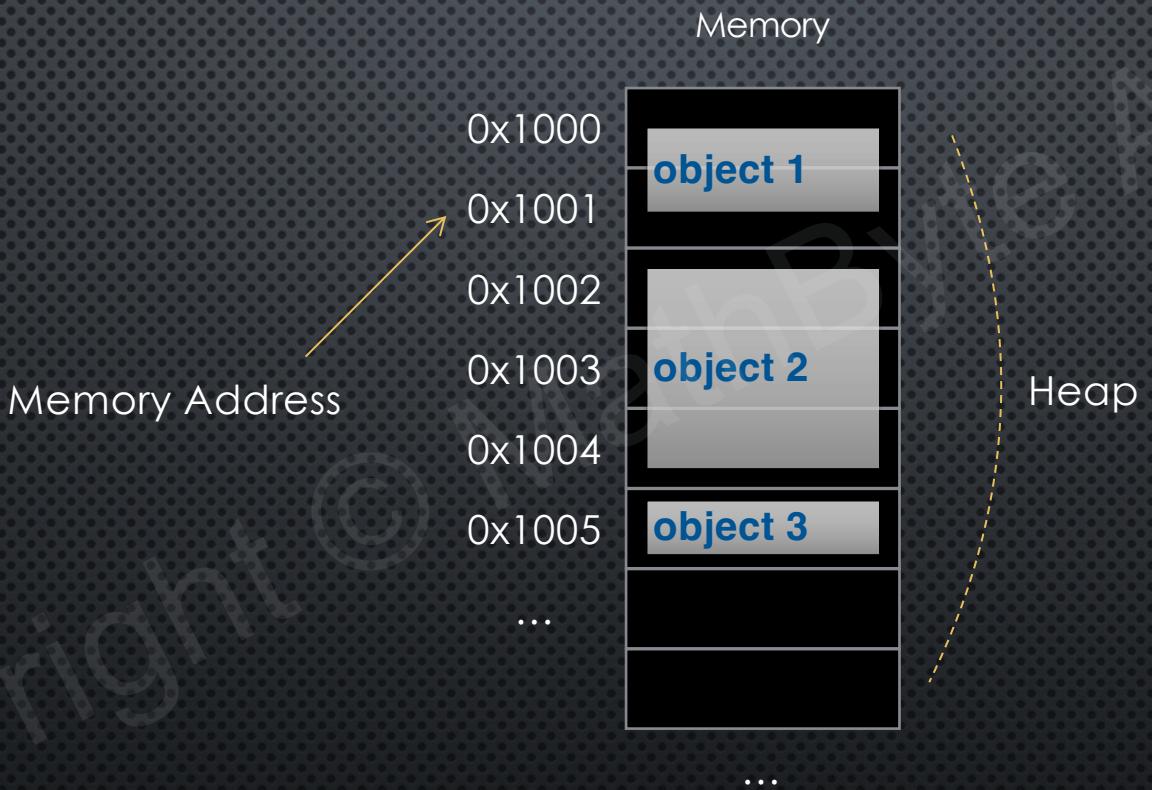
shared references

variable equality

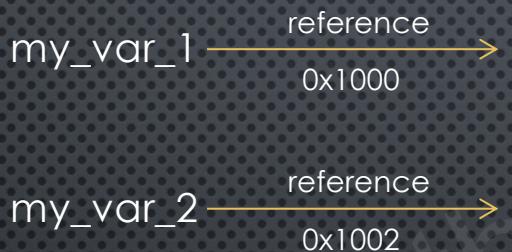
everything is an object

VARIABLES ARE MEMORY REFERENCES

Python Memory Manager



```
my_var_1 = 10
```



```
my_var_2 = 'hello'
```

my_var_1 **references** the object at 0x1000

my_var_2 **references** the object at 0x1002

Memory

10
hello
...

...

In Python, we can find out the memory address referenced by a variable by using the **id()** function. This will return a base-10 number. We can convert this base-10 number to hexadecimal, by using the **hex()** function.

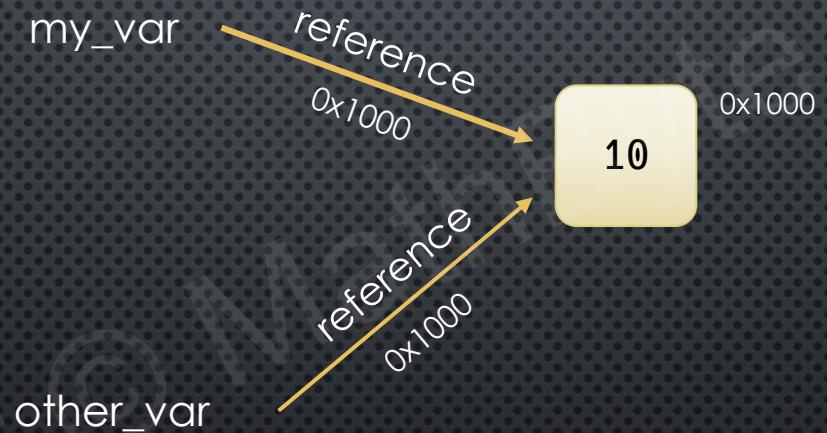
Example

```
a = 10  
print(hex(id(a)))
```

REFERENCE COUNTING

```
my_var = 10
```

```
other_var = my_var
```



reference	count
0x1000	1 2 1 0

Reference Counting
Python Memory Manager

Finding the Reference Count

```
sys.getrefcount(my_var)
```

← passing `my_var` to `getrefcount()` creates an extra reference!

```
ctypes.c_long.from_address(address).value
```

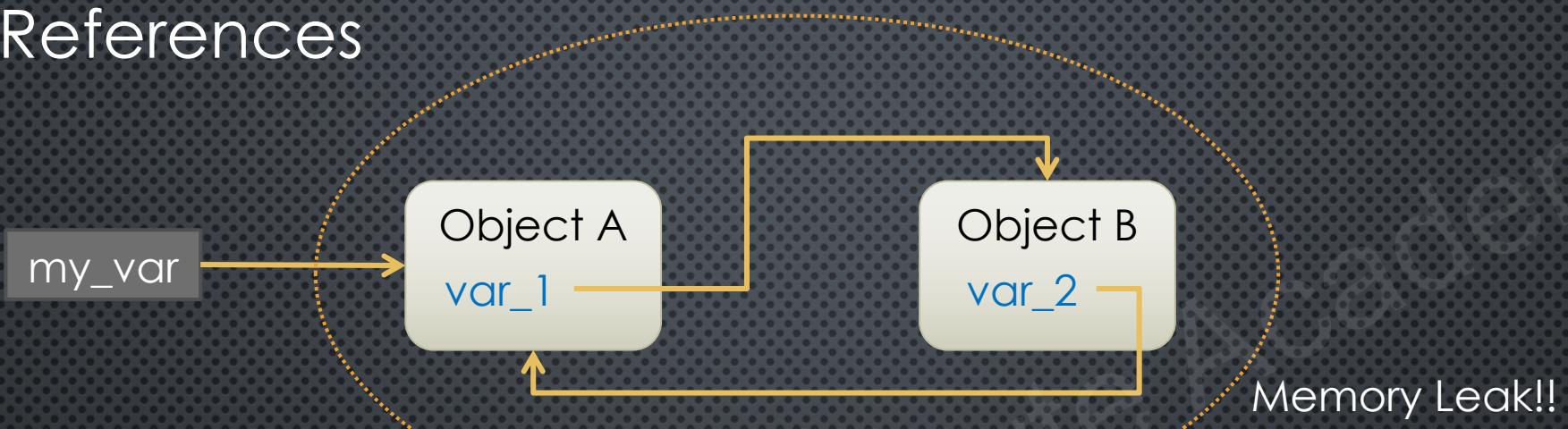


Here, we just pass the memory address (an `integer`), not a reference – does not affect reference count

GARBAGE COLLECTION

Copyright © 2014

Circular References



Garbage Collector

- can be controlled programmatically using the `gc` module
- by default it is turned **on**
- you may turn it **off** if you're **sure** your code does not create circular references – but **beware!!**
- runs periodically on its own (if turned on)
- you can call it manually, and even do your own cleanup

In general GC works just fine
but, not always...

for Python < 3.4

If **even one** of the objects in the circular reference has a destructor [e.g. `__del__()`]

the destruction **order** of the objects may be important

but the GC does not know what that order should be

so the object is marked as **uncollectable**

and the objects in the circular reference are not cleaned up → memory leak

DYNAMIC TYPING VS STATIC TYPING

Some languages (Java, C++, Swift) are **statically typed**

```
String myVar = "hello";
```

data type	variable name	value
String	myVar	"hello"



```
myVar = 10;
```

Does not work!

myVar has been **declared** as a String, and cannot be assigned the integer value 10 later.

```
myVar = "abc";
```

This is OK!

"abc" is a String – so compatible type and assignment works.

Python, in contrast, is **dynamically** typed.

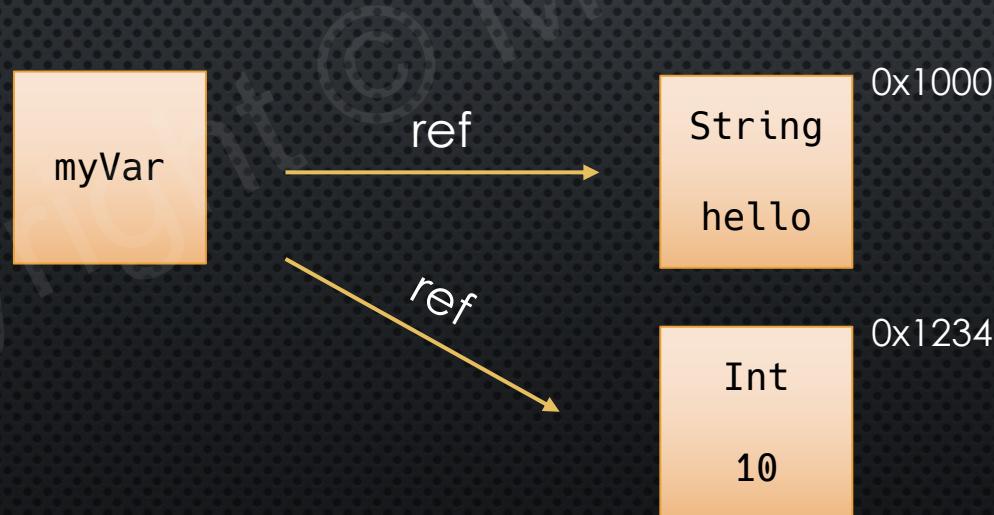
```
my_var = 'hello';
```

The variable `my_var` is purely a **reference** to a string object with value `hello`.

No type is “attached” to `my_var`.

```
my_var = 10;
```

The variable `my_var` is now pointing to an integer object with value `10`.



We can use the built-in `type()` function to determine the type of the object currently referenced by a variable.

Remember: variables in Python do not have an inherent static type.

Instead, when we call `type(my_var)`

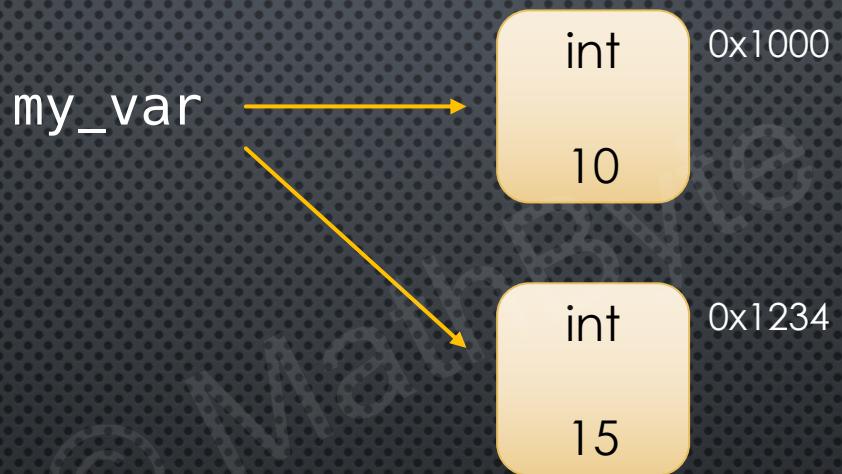
Python looks up the object my_var is referencing (pointing to), and returns the type of the object at that memory location.

VARIABLE RE-ASSIGNMENT

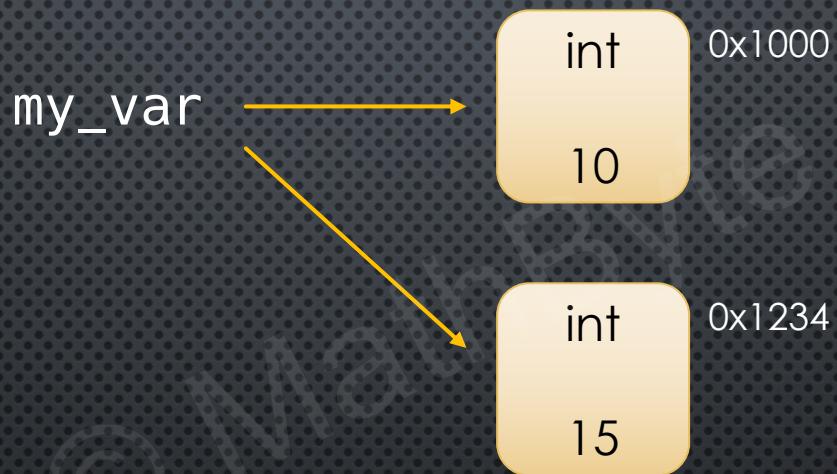
Copyright © 2014

```
my_var = 10
```

```
my_var = 15
```



```
my_var = 10
```



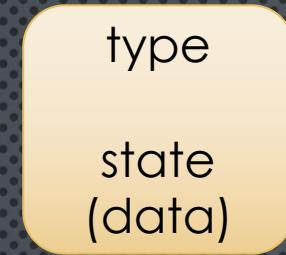
```
my_var = my_var + 5
```

In fact, the value inside the int objects, can **never** be changed!

OBJECT MUTABILITY

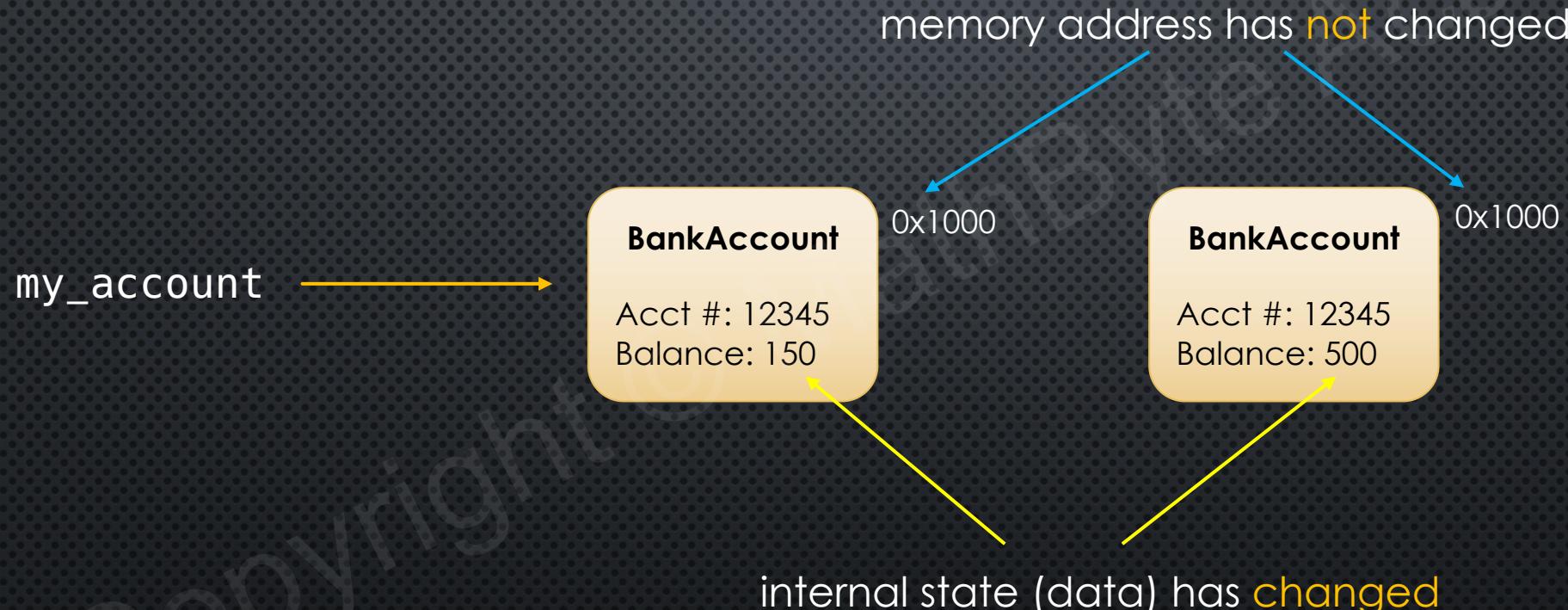
Copyright © 2018

Consider an object in memory:



0x1000

Changing the data **inside** the object is called **modifying the internal state** of the object.



Object was mutated

→ fancy way of saying the internal data has changed

An object whose internal state **can** be changed, is called

Mutable

An object whose internal state **cannot** be changed, is called

Immutable

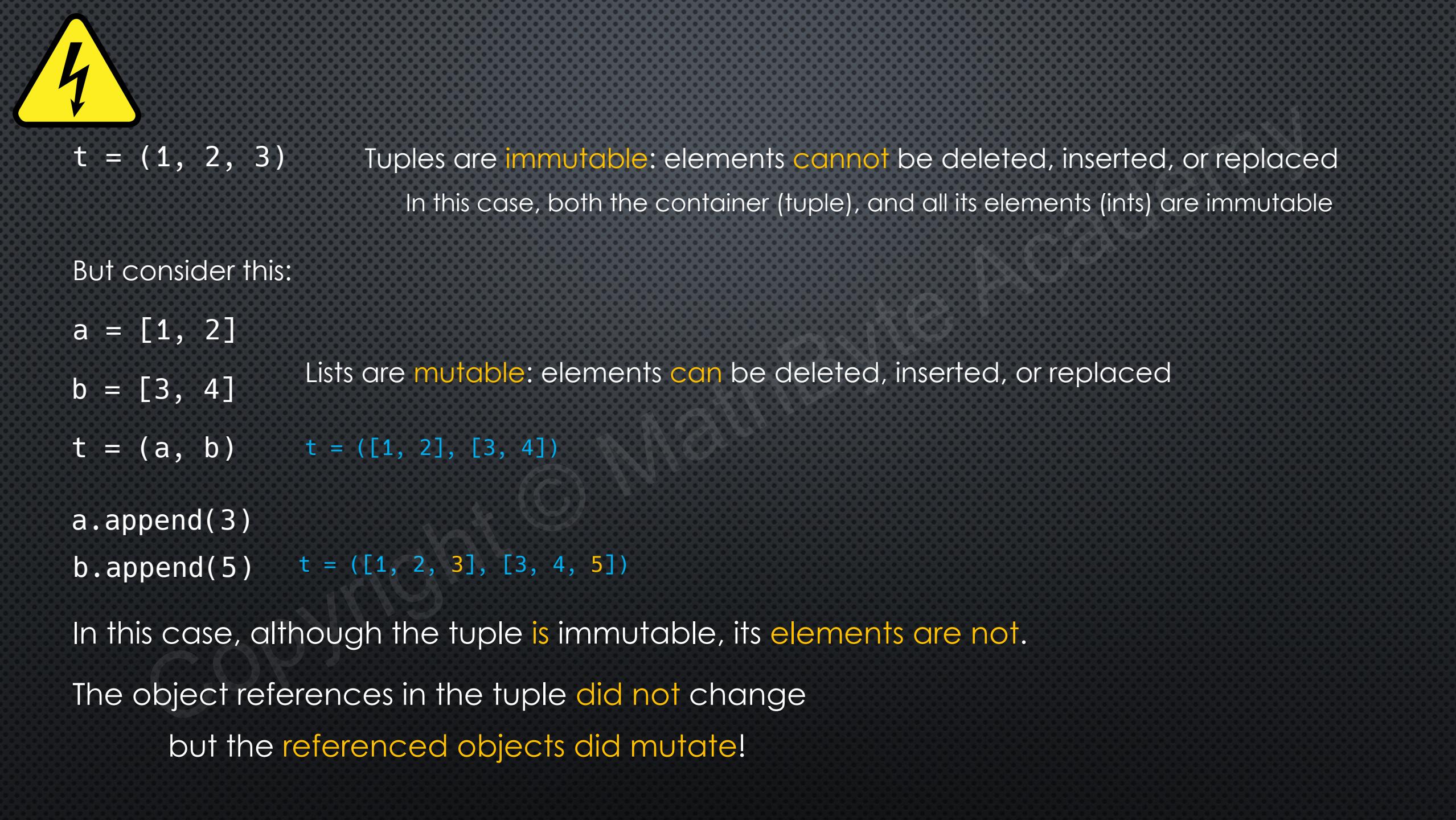
Examples in Python

Immutable

- Numbers (int, float, Booleans, etc)
- Strings
- Tuples
- Frozen Sets
- User-Defined Classes

Mutable

- Lists
- Sets
- Dictionaries
- User-Defined Classes



```
t = (1, 2, 3)
```

Tuples are **immutable**: elements **cannot** be deleted, inserted, or replaced
In this case, both the container (tuple), and all its elements (ints) are immutable

But consider this:

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

```
b = [3, 4]
```

```
t = (a, b)    t = ([1, 2], [3, 4])
```

```
a.append(3)
```

```
b.append(5)    t = ([1, 2, 3], [3, 4, 5])
```

In this case, although the tuple **is** immutable, its elements **are not**.

The object references in the tuple **did not** change
but the **referenced objects did mutate!**

```
t = (1, 2, 3)
```

tuple is immutable



these are references to immutable object (int)

```
t = ([1, 2], [3, 4])
```

tuple is immutable



these are references to a mutable object (list)

FUNCTION ARGUMENTS AND MUTABILITY

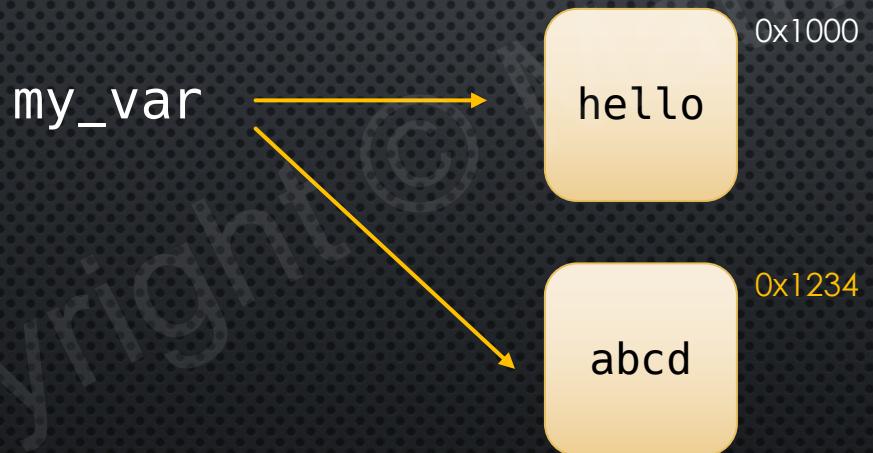
Copyright © 2018

In Python, Strings (**str**) are **immutable** objects.

Once a string has been created, the contents of the object can **never** be changed

In this code: `my_var = 'hello'`

the only way to modify the “value” of `my_var` is to re-assign `my_var` to **another** object



Immutable objects are safe from unintended side-effects

but watch out for immutable collection objects that contain mutable objects

```
def process(s):  
    s = s + ' world'  
    return s  
  
my_var = 'hello'  
  
process(my_var)  
  
print(my_var) -----> hello
```

my_var's reference is passed to process()

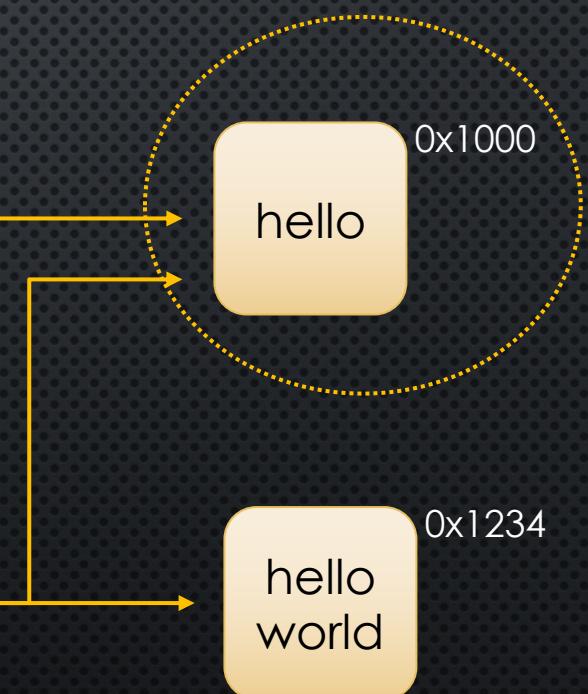
Scopes

module scope

my_var

process() scope

s



Mutable objects are **not** safe from unintended side-effects

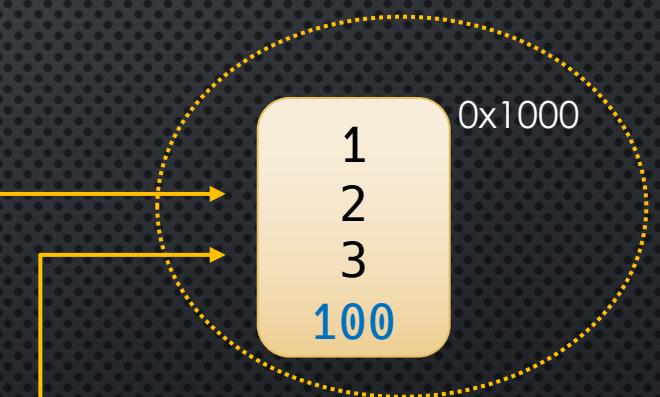
```
def process(lst):  
    lst.append(100)  
  
my_list = [1, 2, 3]  
  
process(my_list)  
  
print(my_list)  
-----> [1, 2, 3, 100]
```

my_list's reference is passed to process()

Scopes

module scope
my_list

process() scope
lst



Immutable collection objects that contain mutable objects

```
def process(t):  
    t[0].append(3)
```

```
my_tuple = ([1,2], 'a')
```

```
process(my_tuple)
```

```
print(my_tuple)
```

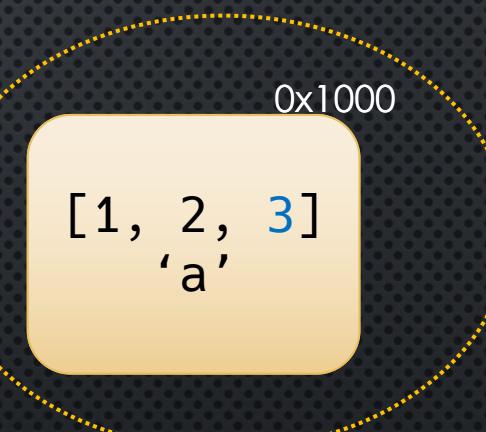
```
-----> ([1, 2, 3], 'a')
```

my_tuple's reference is passed to process()

Scopes

module scope
my_tuple

process() scope
t

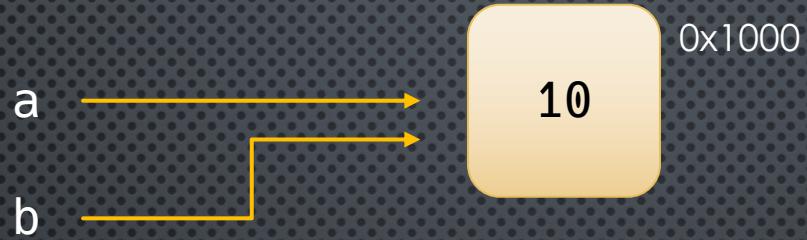


SHARED REFERENCES AND MUTABILITY

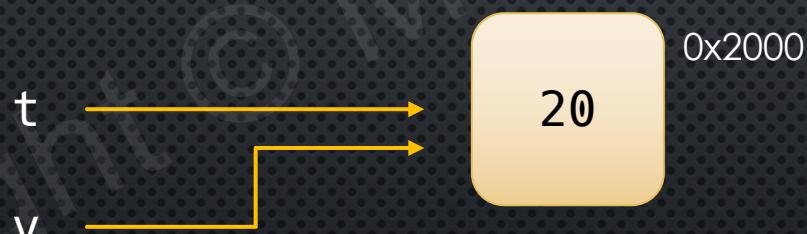
Copyright © 2024

The term **shared reference** is the concept of two variables referencing the **same** object in memory (i.e. having the same memory address)

```
a = 10  
b = a
```

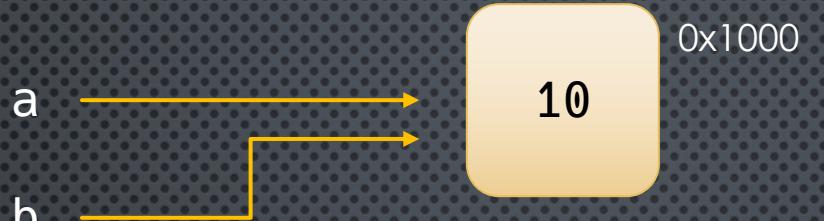


```
def my_func(v):  
    ...  
t = 20  
my_func(t)
```

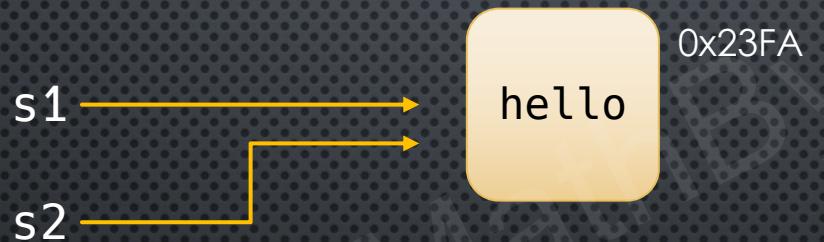


In fact, the following may surprise you:

```
a = 10  
b = 10
```



```
s1 = 'hello'  
s2 = 'hello'
```



In both these cases, Python's memory manager decides to automatically re-use the memory references!!

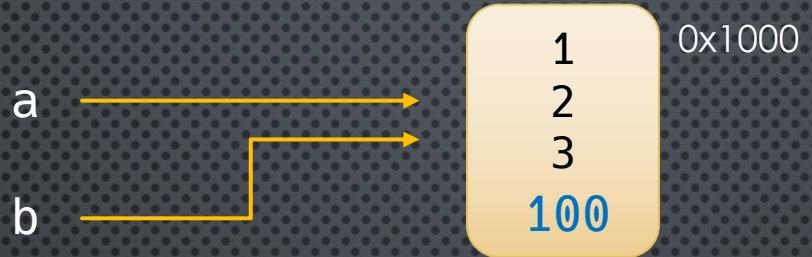
We'll revisit this again soon

Is this even safe? Yes

The integer object 10, and the string object 'hello' are immutable – so it is safe to set up a shared reference

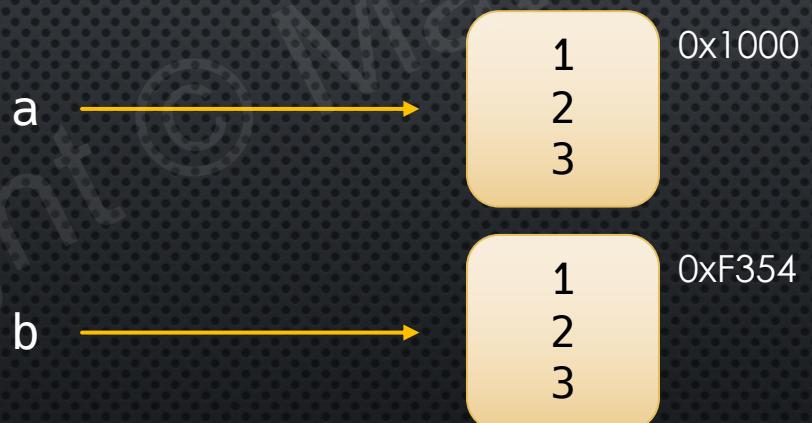
When working with **mutable** objects we have to be more careful

```
a = [1, 2, 3]  
b = a  
  
b.append(100)
```



With mutable objects, the Python memory manager will never create shared references

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



VARIABLE EQUALITY

Copyright © 2014
NCC Group

We can think of variable equality in two fundamental ways:

Memory Address

is

identity operator

`var_1 is var_2`

Object State (data)

==

equality operator

`var_1 == var_2`

Negation

is not

`var_1 is not var_2`

!=

`var_1 != var_2`

`not(var_1 is var_2)`

`not(var_1 == var_2)`

Examples

```
a = 10  
b = a
```

a is b 
a == b 

```
a = 'hello'  
b = 'hello'
```

a is b 
a == b 

but as we'll see later, don't count on it!

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

a is b 
a == b 

```
a = 10  
b = 10.0
```

a is b 
a == b 

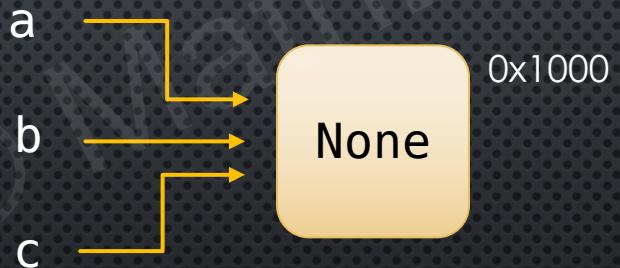
The `None` object

The `None` object can be assigned to variables to indicate that they are not set (in the way we would expect them to be), i.e. an “empty” value (or null pointer)

But the `None` object is a `real` object that is managed by the Python memory manager

Furthermore, the memory manager will always use a `shared reference` when assigning a variable to `None`

```
a = None  
b = None  
c = None
```



So we can test if a variable is “not set” or “empty” by comparing it’s memory address to the memory address of `None` using the `is` operator

`a is None`

`x = 10`

`x is None`

`x is not None`

EVERYTHING IS AN OBJECT

Copyright © 2014

Throughout this course, we'll encounter many data types.

- Integers (`int`)
- Booleans (`bool`)
- Floats (`float`)
- Strings (`str`)
- Lists (`list`)
- Tuples (`tuple`)
- Sets (`set`)
- Dictionaries (`dict`)
- None (`NoneType`)

We'll also see other constructs:

- Operators (`+`, `-`, `==`, `is`, ...)
- Functions
- Classes
- Types

and many more...

But the one thing in common with all these things, is that they are all **objects** (instances of classes)

- Functions (**function**) ←
- Classes (**class**) [not just instances, but the class itself]
- Types (**type**)

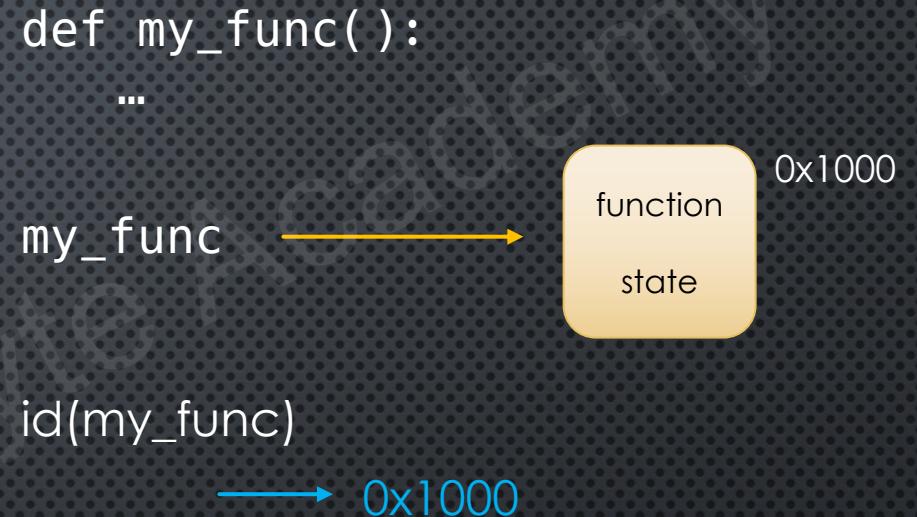
This means they all have a **memory address**!

As a consequence:

Any object can be assigned to a variable
including functions...

Any object can be passed to a function
including functions...

Any object can be returned from a function
including functions...



`my_func` is the **name** of the function
`my_func()` invokes the function

PYTHON OPTIMIZATIONS

INTERNING

Important Note:

A lot of what we discuss with memory management, garbage collection and optimizations, is usually specific to the Python implementation you use.

In this course, we are using **C**Python, the standard (or reference) Python implementation (written in C).

But there are other Python implementations out there. These include:

- **Jython** – written in Java and can import and use any Java class – in fact it even compiles to Java bytecode which can then run in a JVM
- **IronPython** – this one is written in C# and targets .Net (and mono) CLR
- **PyPy** – this one is written in RPython (which is itself a statically-typed subset of Python written in C that is specifically designed to write interpreters)
- and many more...

<https://wiki.python.org/moin/PythonImplementations>

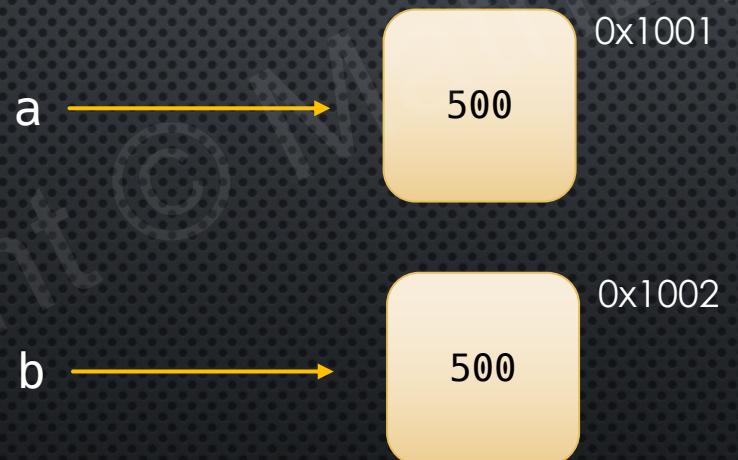
Earlier we saw:

```
a = 10  
b = 10
```



But look at this:

```
a = 500  
b = 500
```



In this case, although it would be safe for Python to create a shared reference, it does not!

What is going on?

Interning: reusing objects on-demand

At startup, Python (CPython), **pre-loads** (caches) a global list of integers in the range [-5, 256]

Any time an integer is referenced in that range, Python will use the cached version of that object

Singletons

Optimization strategy – small integers show up often

When we write

`a = 10`

Python just has to point to the existing reference for 10

But if we write

`a = 257`

Python does not use that global list and a new object
is created every time

PYTHON OPTIMIZATIONS

STRING INTERNING

Some strings are also automatically **interned** – but not all!

As the Python code is compiled, **identifiers** are interned

- variable names
- function names
- class names
- etc.

Identifiers:

- must start with _ or a letter
- can only contain _, letters and numbers

Some string literals may also be automatically interned:

- string literals that look like identifiers (e.g. 'hello_world')
- although if it starts with a digit, even though that is not a valid identifier, it may still get interned

But don't count on it!!

Why do this?

It's all about (speed and, possibly, memory) optimization.

Python, both internally, and in the code you write, deals with lots and lots of dictionary type lookups, on string keys, which means a lot of **string equality** testing.

Let's say we want to see if two strings are equal:

```
a = 'some_long_string'    b = 'some_long_string'
```

Using `a == b`, we need to compare the two strings **character by character**

But if we know that '`'some_long_string'`' has been **interned**, then `a` and `b` are the same string if they both point to the **same memory address**

In which case we can use `a is b` instead – which compares two **integers** (memory address)

This is **much** faster than the character by character comparison

Not all strings are automatically interned by Python

But you can force strings to be interned by using the `sys.intern()` method.

```
import sys  
  
a = sys.intern('the quick brown fox')  
b = sys.intern('the quick brown fox')
```

`a is b` → True

much faster than `a == b`

When should you do this?

- dealing with a large number of strings that could have high repetition
e.g. tokenizing a large corpus of text (NLP)
- lots of string comparisons

In general though, you do not need to intern strings yourself. Only do this if you really need to.

PYTHON OPTIMIZATIONS

PEEPHOLE

This is another variety of optimizations that can occur at compile time.

Constant expressions

numeric calculations

`24 * 60`

Python will actually pre-calculate `24 * 60 → 1440`

short sequences length < 20

`(1, 2) * 5` → `(1, 2, 1, 2, 1, 2, 1, 2)`

`'abc' * 3` → `abcabcabc`

`'hello' + ' world'` → `hello world`

but not `'the quick brown fox' * 10`

(more than 20 characters)

Membership Tests: Mutables are replaced by Immutables

When membership tests such as:

```
if e in [1, 2, 3]:
```

are encountered, the [1, 2, 3] constant, is replaced by its immutable counterpart

(1, 2, 3) tuple

- lists → tuples
- sets → frozensets

Set membership is much faster than list or tuple membership (sets are basically like dictionaries)

So, instead of writing:

```
if e in [1, 2, 3]: or if e in (1, 2, 3):
```

write if e in {1, 2, 3}:

NUMBERS

Copyright © 2014 McGraw-Hill Education

~~5~~

Four main types of numbers:

Boolean truth values

0 (`False`), 1 (`True`)

Python Type

`bool`

Integer Numbers (\mathbb{Z})

0, $\pm 1, \pm 2, \pm 3, \dots$

`int`

Rational Numbers (\mathbb{Q})

$$\left\{ \frac{p}{q} \mid p, q \in \mathbb{Z}, q \neq 0 \right\}$$

`fractions.Fraction`

Real Numbers (\mathbb{R})

$$0, -1, 0.125, \frac{1}{3}, \pi, \dots$$

`float`

`decimal.Decimal`

Complex Numbers (\mathbb{C})

$$\{ a + bi \mid a, b \in \mathbb{R} \}$$

`complex`

$$\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

INTEGERS

DATA TYPE

The `int` data type

Ex: 0, 10, -100, 1000000000, ...

How large can a Python `int` become (positive or negative)?

Integers are represented internally using base-2 (binary) digits, not decimal.

$$(10011)_2 = (19)_{10}$$

Representing the decimal number 19 requires **5 bits**

What's the largest (base 10) integer number that can be represented using 8 bits?

Let's assume first that we only care about non-negative integers

$$\begin{array}{cccccccc} \underline{1} & \underline{1} \\ \hline 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \end{array} \quad \begin{aligned} & 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 \\ & = 255 \\ & = 2^8 - 1 \end{aligned}$$

If we care about handling negative integers as well, then 1 bit is reserved to represent the sign of the number, leaving us with only 7 bits for the number itself out of the original 8 bits

The largest number we can represent using 7 bits is $2^7 - 1 = 127$

So, using 8 bits we are able to represent all the integers in the range $[-127, 127]$

Since 0 does not require a sign, we can squeeze out an extra number, and we end up with the range $[-128, 127]$

$[-2^7, 2^7 - 1]$

If we want to use **16** bits to store (signed) integers, our range would be:

$$2^{(16-1)} = 2^{15} = 32,768 \quad \text{Range: } [-32,768 \dots 32,767]$$

Similarly, if we want to use **32** bits to store (signed) integers, our range would be:

$$2^{(32-1)} = 2^{31} = 2,147,483,648 \quad \text{Range: } [-2,147,483,648 \dots 2,147,483,647]$$

If we had an unsigned integer type, using 32 bits our range would be:

$$[0, 2^{32}] = [0 \dots 4,294,967,296]$$

In a 32-bit OS:

memory spaces (bytes) are limited by their address number → 32 bits

4,294,967,296 bytes of addressable memory

$$= 4,294,967,296 / 1024 \text{ kB} = 4,194,304 \text{ kB}$$

$$= 4,194,304 / 1024 \text{ MB} = 4,096 \text{ MB}$$

$$= 4,096 / 1024 \text{ GB} = 4 \text{ GB}$$

So, how large an integer can be depends on how many bits are used to store the number.

Some languages (such as Java, C, ...) provide multiple distinct integer data types that use a fixed number of bits:

Java

<code>byte</code>	signed 8-bit numbers	$-128, \dots, 127$
<code>short</code>	signed 16-bit numbers	$-32,768, \dots, 32,767$
<code>int</code>	signed 32-bit numbers	$-2^{31} \dots, 2^{31} - 1$
<code>long</code>	signed 64-bit numbers	$-2^{63} \dots, 2^{63} - 1$

and more...

Python does not work this way

The `int` object uses a **variable** number of bits

Can use 4 bytes (32 bits), 8 bytes (64 bits), 12 bytes (96 bits), etc.

Seamless to us

[since `int`s are actually objects, there is a further fixed overhead per integer]

Theoretically limited only by the amount of memory available

Of course, larger numbers will use more memory
and standard operators such as `+`, `*`, etc. will run
slower as numbers get larger

INTEGERS

OPERATIONS

Integers support all the standard arithmetic operations:

addition	+
subtraction	-
multiplication	*
division	/
exponents	**

But what is the resulting type of each operation?

`int + int → int`

`int - int → int`

`int * int → int`

`int ** int → int`

`int / int → float`

obviously $3 / 4 \rightarrow 0.75$ (float)
but, also $10 / 2 \rightarrow 5$ (float)

Two more operators in integer arithmetic

First, we revisit long integer division...

$$\begin{array}{r} 155 \text{ // } 4 \\ \hline 38 \\ 4 \overline{)155} \\ 12 \\ \hline 35 \\ 32 \\ \hline 3 \\ \hline 155 \% 4 \end{array}$$

$$155 \div 4 = 38 \text{ with remainder } 3$$

put another way:

$$155 = 4 * 38 + 3$$

$$\begin{aligned} 155 &= 4 * (155 \text{ // } 4) + (155 \% 4) \\ &= 4 * 38 + 3 \end{aligned}$$

// is called floor division (div)

% is called the modulo operator (mod)

and they always satisfy:

$$n = d * (n \text{ // } d) + (n \% d)$$

$$\frac{155}{4} \quad \frac{\text{numerator}}{\text{denominator}}$$

What is floor division exactly?

First define the floor of a (real) number

The floor of a real number a is the largest (in the standard number order) integer $\leq a$

$$\text{floor}(3.14) \rightarrow 3$$

$$\text{floor}(1.9999) \rightarrow 1$$

$$\text{floor}(2) \rightarrow 2$$

But watch out for negative numbers!

$$\text{floor}(-3.1) \rightarrow -4$$



So, floor is not quite the same as truncation!

$$a // b = \text{floor}(a / b)$$

$$a = b * (a // b) + a \% b$$

a = 135

b = 4 $135 / 4 = 33.75$ ($33 \frac{3}{4}$)

$135 // 4 \rightarrow 33$

$135 \% 4 \rightarrow 3$

And, in fact: $a = b * (a // b) + a \% b$

$$\begin{aligned} & 4 * (135 // 4) + (135 \% 4) \\ &= 4 * 33 + 3 \\ &= 132 + 3 \\ &= 135 \end{aligned}$$



Negative Numbers

Be careful, $a // b$, is not the integer portion of a / b , it is the floor of a / b

For $a > 0$ and $b > 0$, these are indeed the same thing

But beware when dealing with negative numbers!

$$a = -135$$

$$b = 4 \quad -135 / 4 = -33.75 (-33 \frac{3}{4})$$

$$-135 // 4 \rightarrow -34$$

$$135 // 4 \rightarrow 33$$

$$-135 \% 4 \rightarrow 1$$

$$135 \% 4 \rightarrow 3$$

And, in fact:

$$a = b * (a // b) + a \% b$$

$$\begin{aligned} & 4 * (-135 // 4) + (-135 \% 4) \\ &= (4 * -34) + 1 \\ &= -136 + 1 \\ &= -135 \end{aligned}$$



Expanding this further...

$a = 13$	$b = 4$	$a = -13$	$b = 4$	$a = 13$	$b = -4$	$a = -13$	$b = -4$
$13 / 4 \rightarrow 3.25$		$-13 / 4 \rightarrow -3.25$		$13 / -4 \rightarrow -3.25$		$-13 / -4 \rightarrow 3.25$	
$13 // 4 \rightarrow 3$		$-13 // 4 \rightarrow -4$		$13 // -4 \rightarrow -4$		$-13 // -4 \rightarrow 3$	
$13 \% 4 \rightarrow 1$		$-13 \% 4 \rightarrow 3$		$13 \% -4 \rightarrow -3$		$-13 \% -4 \rightarrow -1$	

In each of these cases: $a = b * (a // b) + a \% b$

$4 * (13 // 4) + 13 \% 4$ $= 12 + 1 = 13 \checkmark$	$4 * (-13 // 4) + -13 \% 4$ $= -16 + 3 = -13 \checkmark$	$-4 * (13 // -4) + 13 \% -4$ $= 16 + -3 = 13 \checkmark$	$-4 * (-13 // -4) + -13 \% -4$ $= -12 + -1 = -13 \checkmark$
---	---	---	---

INTEGERS

CONSTRUCTORS AND BASES

(PART 1)

An integer number is an object – an instance of the `int` class

The `int` class provides multiple constructors

```
a = int(10)
```

```
a = int(-10)
```

Other (numerical) data types are also supported in the argument of the `int` constructor:

```
a = int(10.9)           -----> truncation: a → 10
```

```
a = int(-10.9)          -----> truncation: a → -10
```

```
a = int(True)           -----> a → 1
```

```
a = int(Decimal("10.9")) -----> truncation: a → 10
```

As well as **strings** (that can be parsed to a number)

```
a = int("10")           -----> a → 10
```

Number Base

`int("123") → (123)10`

When used with a string, constructor has an optional second parameter: `base` $2 \leq \text{base} \leq 36$

If base is not specified, the default is base 10 – as in the example above

`int("1010", 2) → (10)10` `int("1010", base=2) → (10)10`

`int("A12F", base=16) → (41263)10` `int("534", base=8) → (348)10`

`int("a12f", base=16) → (41263)10` `int("A", base=11) → (10)10`

`int("B", 11)` **ValueError:** invalid literal for int() with base 11: 'B'

Reverse Process: changing an integer from base 10 to another base

built-in functions: `bin()` `bin(10) → '0b1010'`

`oct()` `oct(10) → '0o12'`

`hex()` `hex(10) → '0xa'`

The prefixes in the strings help document the base of the number `int('0xA', 16) → (10)10`

These prefixes are consistent with literal integers using a base prefix (no strings attached!)

`a = 0b1010` `a → 10`

`a = 0o12` `a → 10`

`a = 0xA` `a → 10`

What about other bases? Custom code

n: number (base 10)

b: base (target base)

$$\begin{array}{r} \underline{\text{?}} & \underline{\text{?}} \\ b^7 & b^6 & b^5 & b^4 & b^3 & b^2 & b^1 & b^0 \end{array}$$

$$n = b * (n // b) + n \% b$$

$$\rightarrow n = (n // b) * b + n \% b$$

$n = 232$

$b = 5$

$$232 = (232 // 5) \times 5 + 232 \% 5 = 46 \times 5 + 2$$

$$= [46 \times 5^1] + [2 \times 5^0]$$

$$= [(46 // 5) \times 5 + 46 \% 5] \times 5^1 + [2 \times 5^0]$$

$$= [(9 \times 5 + 1) \times 5^1] + [2 \times 5^0]$$

$$= [9 \times 5^2] + [1 \times 5^1] + [2 \times 5^0]$$

$$= [(9 // 5) \times 5 + 9 \% 5] \times 5^2 + [1 \times 5^1] + [2 \times 5^0]$$

$$= [(1 \times 5 + 4) \times 5^2] + [1 \times 5^1] + [2 \times 5^0]$$

$$= [1 \times 5^3] + [4 \times 5^2] + [1 \times 5^1] + [2 \times 5^0]$$

div 3rd mod 2nd mod 1st mod

$$= [(1 // 5) \times 5 + 1 \% 5] \times 5^3 + [4 \times 5^2] + [1 \times 5^1] + [2 \times 5^0]$$

$$= [(0 \times 5 + 1) \times 5^3] + [4 \times 5^2] + [1 \times 5^1] + [2 \times 5^0]$$

$$= [0 \times 5^4] + [1 \times 5^3] + [4 \times 5^2] + [1 \times 5^1] + [2 \times 5^0]$$

stop 4th mod 3rd mod 2nd mod 1st mod

$$\begin{array}{r} ? \\ \hline 5^3 & 5^2 & 5^1 & 5^0 \end{array}$$

$$\begin{array}{r} ? \\ \hline 5^3 & 5^2 & 46 & 2 \\ \hline 5^1 & 5^0 \end{array}$$

$$\begin{array}{r} ? \\ \hline 5^3 & 9 & 1 & 2 \\ \hline 5^2 & 5^1 & 5^0 \end{array}$$

$$\begin{array}{r} 1 \\ \hline 5^3 & 5^2 & 5^1 & 5^0 \end{array}$$

$$\begin{array}{r} 1 \\ \hline 5^3 & 5^2 & 5^1 & 5^0 \end{array}$$

too big

too big

Base Change Algorithm

```
n = base-10 number (>= 0)      b = base (>= 2)
```

```
if b < 2 or n < 0: raise exception  
if n == 0: return [0]
```

```
digits = []  
while n > 0:  
    m = n % b  
    n = n // b  
    digits.insert(0, m)
```

n = 232 b = 5 digits → [1, 4, 1, 2]

n = 1485 b = 16 digits → [5, 12, 13]

This algorithm returns a list of the digits in the specified base b (a representation of n_{10} in base b)

Usually we want to return an encoded number where digits higher than 9 use letters such as A..Z

We simply need to decide what character to use for the various digits in the base.

Encodings

Typically, we use 0-9 and A-Z for digits required in bases higher than 10

But we don't have to use letters or even standard 0-9 digits to encode our number.

We just need to map between the digits in our number, to a character of our choice.

0 → 0	0 → 0	0 → a
1 → 1	1 → 1	1 → b
...
9 → 9	10 → A	9 → i
	11 → B	10 → #
10 → A	...	11 → !
11 → B	37 → a	...
...	38 → b	36 → *
36 → Z	...	
	62 → z	

Python uses 0-9 and a-z (case insensitive)
and is therefore limited to base <= 36

Your choice of characters to represent the digits, is your **encoding map**

Encodings

The simplest way to do this given a list of digits to encode, is to create a string with as many characters as needed, and use their index (ordinal position) for our encoding map

```
base b (>=2)  
map = ' ... ' (of length b)  
digits = [ ... ]  
encoding = map[digits[0]] + map[digits[1]] + ...
```

Example: Base 12

map = '0123456789ABC'

The string '0123456789ABC' is shown. Three yellow arrows point from the numbers 10, 11, and 12 to the characters B, C, and A respectively, indicating their positions in the mapping.

digits = [4, 11, 3, 12]

encoding = '4B3C'

Encoding Algorithm

```
digits = [ ... ]  
map = ' ... '  
  
encoding = ''  
for d in digits:  
    encoding += map[d]      (a += b → a = a + b)
```

or, more simply:

```
encoding = ''.join([map[d] for d in digits])
```

we'll cover this in much more detail in the section on lists

INTEGERS

CONSTRUCTORS AND BASES

(PART 2)

Constructor

`int(a)` `a` is a numeric type such as float, Decimal, Fraction, bool, ...
`int(s, base=10)` `s` is a string, and `base` is the base used for the encoded string `s`

Base Change Algorithm

```
n = base-10 number (>= 0)  
b = base (>= 2)  
  
if b < 2 or n < 0: raise exception  
if n == 0: return [0]  
  
digits = []  
while n > 0:  
    m = n % b  
    n = n // b  
    digits.insert(0, m)
```

Encoding Algorithm

```
digits = [...]  
map = '...'  
  
encoding = ''  
for d in digits:  
    encoding += map[d]
```

or, more simply:

```
encoding = ''.join([map[d] for d in digits])
```

RATIONAL NUMBERS

Copyright ©  NCES

Rational numbers are **fractions** of integer numbers

Ex: $\frac{1}{2}$ $-\frac{22}{7}$

Any real number with a **finite** number of digits after the decimal point is **also** a rational number

$$0.45 \rightarrow \frac{45}{100}$$

$$0.123456789 \rightarrow \frac{123456789}{10^9}$$

So $\frac{8.3}{4}$ is also rational

$$\frac{8.3}{4} = \frac{83/10}{4} = \frac{83}{10} \times \frac{1}{4} = \frac{83}{40}$$

as is $\frac{8.3}{1.4}$ since

$$\frac{8.3}{1.4} = \frac{83/10}{14/10} = \frac{83}{10} \times \frac{10}{14} = \frac{83}{14}$$

The Fraction Class

Rational numbers can be represented in Python using the `Fraction` class in the `fractions` module

```
from fractions import Fraction  
x = Fraction(3, 4)  
y = Fraction(22, 7)  
z = Fraction(6, 10)
```

Fractions are automatically reduced:

`Fraction(6, 10) → Fraction(3, 5)`

Negative sign, if any, is always attached to the numerator:

`Fraction(1, -4) → Fraction(-1, 4)`

Standard arithmetic operators are supported: +, -, *, /
and result in **Fraction** objects as well

$$\frac{2}{3} \times \frac{1}{2} = \frac{2}{6} = \frac{1}{3}$$

`Fraction(2, 3) * Fraction(1, 2) → Fraction(1, 3)`

$$\frac{2}{3} + \frac{1}{2} = \frac{4}{6} + \frac{3}{6} = \frac{7}{6}$$

`Fraction(2, 3) + Fraction(1, 2) → Fraction(7, 6)`

getting the **numerator** and **denominator** of **Fraction** objects:

`x = Fraction(22, 7)`

`x.numerator` → 22

`x.denominator` → 7

`float` objects have `finite` precision \Rightarrow any `float` object can be written as a fraction!

`Fraction(0.75)` \rightarrow `Fraction(3, 4)`

`Fraction(1.375)` \rightarrow `Fraction(11, 8)`

```
import math
```

```
x = Fraction(math.pi)      → Fraction(884279719003555, 281474976710656)
```

```
y = Fraction(math.sqrt(2)) → Fraction(6369051672525773, 4503599627370496)
```

Even though π and $\sqrt{2}$ are both irrational

internally represented as floats

\Rightarrow finite precision real number

\Rightarrow expressible as a rational number

but it is an approximation



Converting a `float` to a `Fraction` has an important caveat

We'll examine *this* in detail in a later video on `floats`

$\frac{1}{8}$ has an exact float representation

`Fraction(0.125)` → `Fraction(1, 8)`

$\frac{3}{10}$ does not have an exact float representation

`Fraction(0.3)` → `Fraction(5404319552844595, 18014398509481984)`

`format(0.3, '.5f')` → `0.30000`

`format(0.3, '.25f')` → `0.299999999999999888977698`

Constraining the denominator

Given a `Fraction` object, we can find an approximate equivalent fraction with a constrained denominator

using the `limit_denominator(max_denominator=1000000)` instance method

i.e. finds the closest rational (which could be precisely equal)
with a denominator that does not exceed `max_denominator`

```
x = Fraction(math.pi)      → Fraction(884279719003555, 281474976710656)  
                                3.141592653589793
```

```
x.limit_denominator(10)    → Fraction(22, 7)  
                                3.142857142857143
```

```
x.limit_denominator(100)   → Fraction(311, 99)  
                                3.141414141414141
```

```
x.limit_denominator(500)   → Fraction(355, 113)  
                                3.141592920353983
```

FLOATS

INTERNAL REPRESENTATION

The float class is Python's default implementation for representing real numbers

The Python (CPython) float is implemented using the C double type which (usually!) implements the IEEE 754 double-precision binary float, also called binary64

The float uses a fixed number of bytes → 8 bytes (but Python objects have some overhead too)
→ 64 bits → 24 bytes (CPython 3.6 64-bit)

These 64 bits are used up as follows:

sign → 1 bit

exponent → 11 bits → range [-1022, 1023] $1.5\text{E}-5 \rightarrow 1.5 \times 10^{-5}$

significant digits → 52 bits → 15-17 significant (base-10) digits

significant digits → for simplicity, all digits except leading and trailing zeros

1.2345 1234.5 12345000000 0.00012345 12345e-50 1.2345e10

Representation: decimal

Numbers can be represented as base-10 integers and fractions:

$$0.75 \rightarrow \frac{7}{10} + \frac{5}{100} \rightarrow 7 \times 10^{-1} + 5 \times 10^{-2}$$

2 significant digits

$$0.256 \rightarrow \frac{2}{10} + \frac{5}{100} + \frac{6}{1000} \rightarrow 2 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3}$$

3 significant digits

$$\begin{aligned} 123.456 &\rightarrow 1 \times 100 + 2 \times 10 + 3 \times 1 + \frac{4}{10} + \frac{5}{100} + \frac{6}{1000} \\ &\rightarrow 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3} \end{aligned}$$

6 significant digits

In general: $d = \sum_{i=-m}^n d_i \times 10^i$

$sign = 0$ for positive
 $sign = 1$ for negative

$$d = (-1)^{sign} \sum_{i=-m}^n d_i \times 10^i$$

Some numbers cannot be represented using a finite number of terms

$$d = (-1)^{sign} \sum_{i=-m}^n d_i \times 10^i$$

Obviously numbers such as

$$\pi = 3.14159 \dots$$

$$\sqrt{2} = 1.4142 \dots$$

but even some rational numbers

$$\begin{aligned}\frac{1}{3} &= 0.33\dot{3} \\ &= \frac{3}{10} + \frac{3}{100} + \frac{3}{1000} + \dots\end{aligned}$$

Representation: binary

Numbers in a computer are represented using bits, not decimal digits

→ instead of powers of 10, we need to use powers of 2

$$\begin{aligned}(0.11)_2 &= \left(\frac{1}{2} + \frac{1}{4}\right)_{10} = (0.5 + 0.25)_{10} = (0.75)_{10} \\ &= (1 \times 2^{-1} + 1 \times 2^{-2})_{10}\end{aligned}$$

Similarly,

$$\begin{aligned}(0.1101)_2 &= \left(\frac{1}{2} + \frac{1}{4} + \frac{0}{8} + \frac{1}{16}\right)_{10} = (0.5 + 0.25 + 0.0625)_{10} = (0.8125)_{10} \\ &= (1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4})_{10}\end{aligned}$$

This representation is very similar to the one we use with decimal numbers

but instead of using powers of 10, we use powers of 2

a **binary** representation

$$d = (-1)^{\text{sign}} \sum_{i=-m}^n d_i \times 2^i$$

The same problem that occurs when trying to represent $\frac{1}{3}$ using a decimal expansion also happens when trying to represent certain numbers using a binary expansion

$$0.1 = \frac{1}{10} \quad \text{Using binary fractions, this number does not have a finite representation}$$

$$(0.1)_{10} = (0.0\ 0011\ 0011\ 0011\ \dots)_2$$

$$\begin{aligned} &= \frac{0}{2} + \frac{0}{4} + \frac{0}{8} + \frac{1}{16} + \frac{1}{32} + \frac{0}{64} + \frac{0}{128} + \frac{1}{256} + \frac{1}{512} + \frac{0}{1024} + \frac{0}{2048} + \frac{1}{4096} + \frac{1}{8192} + \dots \\ &= \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \dots \end{aligned}$$

$$\begin{aligned} &= 0.0625 + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \dots \\ &= 0.09375 + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \dots \end{aligned}$$

$$\begin{aligned} &= 0.09765625 + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \dots \\ &= 0.099609375 + \frac{1}{4096} + \frac{1}{8192} + \dots \end{aligned}$$

$$\begin{aligned} &= 0.0999755859375 + \dots \end{aligned}$$

base 10

So, some numbers that do have a finite decimal representation,
do not have a finite binary representation,
and some do

$$(0.75)_{10} = (0.11)_2$$

finite

$$(0.8125)_{10} = (0.1101)_2$$

finite



exact float representation

$$(0.1)_{10} = (0.0011 0011 0011 \dots)_2$$

infinite



approximate float representation

FLOATS

EQUALITY TESTING

In the previous video we saw that some decimal numbers (with a finite representation) cannot be represented with a finite binary representation

This can lead to some "weirdness" and bugs in our code (but not a Python bug!!)

```
x = 0.1 + 0.1 + 0.1      format(x, '.25f') → 0.3000000000000000444089210
y = 0.3                  format(y, '.25f') → 0.299999999999999888977698
x == y → False
```

Using rounding will not necessarily solve the problem either!

It is no more possible to exactly represent `round(0.1, 1)` than `0.1` itself

```
round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1) → False
```

But it can be used to round the entirety of both sides of the equality comparison

```
round(0.1 + 0.1 + 0.1, 5) == round(0.3, 5) → True
```

To test for "equality" of two different floats, you could do the following methods:

round both sides of the equality expression to the number of significant digits

```
round(a, 5) == round(b, 5)
```

or, more generally, use an appropriate range (ε) within which two numbers are deemed equal

for some $\varepsilon > 0$, $a = b$ if and only if $|a - b| < \varepsilon$

```
def is_equal(x, y, eps)
    return math.fabs(x-y) < eps
```

This can be tweaked by specifying that the difference between the two numbers be a percentage of their size → the smaller the number, the smaller the tolerance

i.e. are two numbers within x% of each other?

But there are non-trivial issues with using these seemingly simple tests

→ numbers very close to zero vs away from zero

Using absolute tolerances...

```
x = 0.1 + 0.1 + 0.1  
y = 0.3  
print(format(x, '.20f')) → 0.30000000000000004441  
print(format(y, '.20f')) → 0.29999999999999998890
```

17th digit after decimal pt

$$\Delta = \begin{matrix} 0.0000000000000005551 \\ 0.0000000000000001 \end{matrix}$$

```
a = 10000.1 + 10000.1 + 10000.1  
b = 30000.3  
print(format(a, '.20f')) → 30000.3000000000291038305  
print(format(b, '.20f')) → 30000.2999999999927240424
```

12th digit after decimal pt

$$\Delta = \begin{matrix} 0.0000000000363797881 \\ 0.0000000000000001 \end{matrix}$$

Using an absolute tolerance: $\text{abs_tol} = 10^{-15} = 0.0000000000000001$

then

```
math.fabs(x - y) < abs_tol → True
```

```
math.fabs(a - b) < abs_tol → False
```

Maybe we should use relative tolerances...

```
x = 0.1 + 0.1 + 0.1
```

```
y = 0.3
```

→ tol = 0.0000030000000000

```
a = 10000.1 + 10000.1 + 10000.1
```

```
b = 30000.3
```

→ tol = 0.3000030000000000

Using a relative tolerance: rel_tol = 0.001% = 0.00001 = 1e-5

i.e. maximum allowed difference between the two numbers,

relative to the larger magnitude of the two numbers

```
tol = rel_tol * max( |x|, |y| )
```

```
math.fabs(x - y) < tol → True
```

```
math.fabs(a - b) < tol → True
```

Success! but is it really?

```
x = 0.0000000001      (1e-10)
y = 0
```

Using a relative tolerance: `rel_tol = 0.1% = 0.0001 = 1e-3`

```
tol = rel_tol * max(|x|, |y|) → tol = rel_tol * |x| → 1e-3 * 1e-10 = 1e-13
```

```
math.fabs(x - y) < abs_tol → False
```

Using a relative tolerance technique does not work well for numbers **close to zero!**

So using absolute and relative tolerances, in isolation, makes it difficult to get a one-size-fits-all solution

We can **combine** both methods
calculating the absolute and relative tolerances
and using the **larger** of the two tolerances

```
tol = max(rel_tol * max(|x|, |y|), abs_tol)
```

The `math` module has that solution for us!

→ PEP 485

```
math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
```



If you do not specify `abs_tol`, then it defaults to `0` and you will face the problem we encountered in the last slide when comparing numbers close to zero.

```
x = 1000.0000001  
y = 1000.0000002
```

```
a = 0.0000001  
b = 0.0000002
```

```
math.isclose(x, y) → True
```

```
math.isclose(a, b) → False
```

but

```
math.isclose(x, y, abs_tol=1e-5) → True
```

```
math.isclose(a, b, abs_tol=1e-5) → True
```

Also works well in situations like this:

```
x = 1000.01  
y = 1000.02
```

```
math.isclose(x, y, rel_tol=1e-5, abs_tol=1e-5) → True
```

```
a = 0.01  
b = 0.02
```

```
math.isclose(x, y, rel_tol=1e-5, abs_tol=1e-5) → False
```

If you are going to be using this method, you should play around with it for a while until you get a good feel for how it works

FLOATS

COERCING TO INTEGERS

Float → Integer

data loss

different ways to configure this data loss

10.4

10.5

10.6

10? 11?

10.0001

10.9999

truncation

floor

ceiling

rounding

data loss in all cases
pick your poison!

Truncation

truncating a float simply returns the integer portion of the number

i.e. ignores everything after the decimal point

The `math` module provides us the `trunc()` function:

```
import math  
  
math.trunc(10.4) → 10  
math.trunc(10.5) → 10  
math.trunc(10.6) → 10  
  
math.trunc(-10.4) → -10  
math.trunc(-10.5) → -10  
math.trunc(-10.6) → -10
```

The `int` Constructor

The Python `int` constructor will accept a `float`

uses truncation when casting the `float` to an `int`

`int(10.4) → 10`

`int(10.5) → 10`

`int(10.6) → 10`

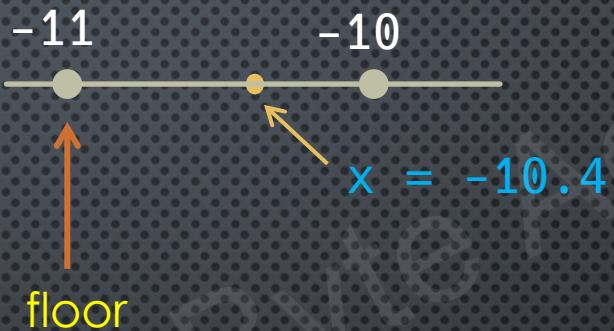
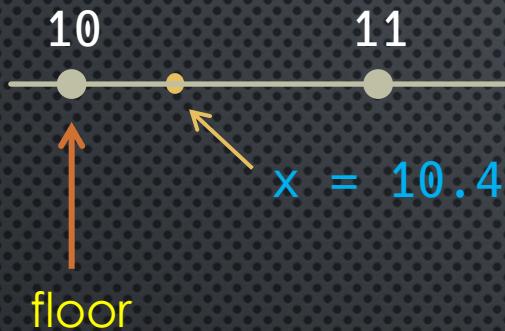
`int(-10.4) → -10`

`int(-10.5) → -10`

`int(-10.6) → -10`

Floor

Definition: the floor of a number is the largest integer less than (or equal to) the number



$$\text{floor}(x) = \max \{i \in \mathbb{Z} \mid i \leq x\}$$

For positive numbers, floor and truncation are equivalent but not for negative numbers!

Recall also our discussion on integer division – aka floor division: //

We defined floor division in combination with the mod operation $n = d * (n // d) + (n \% d)$

But in fact, floor division defined that way yields the same result as taking the floor of the floating point division

$$a // b == \text{floor}(a / b)$$

Floor

The `math` module provides us the `floor()` function:

```
import math
```

```
math.floor(10.4) → 10
```

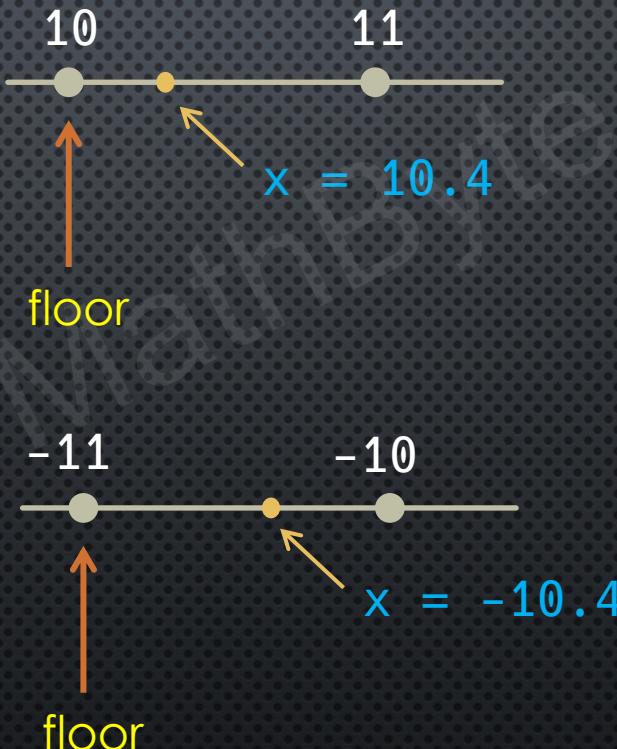
```
math.floor(10.5) → 10
```

```
math.floor(10.6) → 10
```

```
math.floor(-10.4) → -11
```

```
math.floor(-10.5) → -11
```

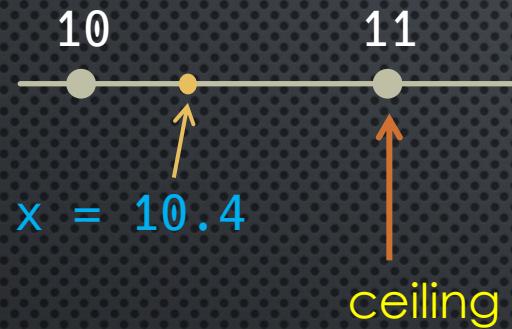
```
math.floor(-10.6) → -11
```



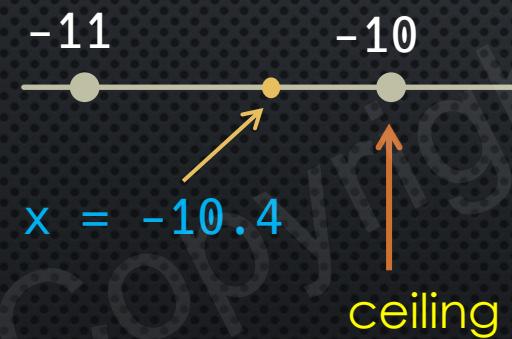
Ceiling

Definition: the ceiling of a number is the smallest integer greater than (or equal to) the number

$$\text{ceil}(x) = \min \{i \in \mathbb{Z} \mid i \geq x\}$$



`math.ceil(10.4) → 11`
`math.ceil(10.5) → 11`
`math.ceil(10.6) → 11`



`math.ceil(-10.4) → -10`
`math.ceil(-10.5) → -10`
`math.ceil(-10.6) → -10`

FLOATS

ROUNDING

The `round()` function

Python provides a built-in rounding function: `round(x, n=0)`

This will round the number `x` to the closest multiple of 10^{-n}

you might think of this as rounding to a certain number of digits after the decimal point
which would work for positive `n`, but `n` can, in fact, also be negative!

In addition to `truncate`, `floor`, and `ceiling`, we can therefore also use rounding (with `n = 0`) to coerce a float to an integer number

If `n` is not specified, then it defaults to zero and `round(x)` will therefore return an `int`

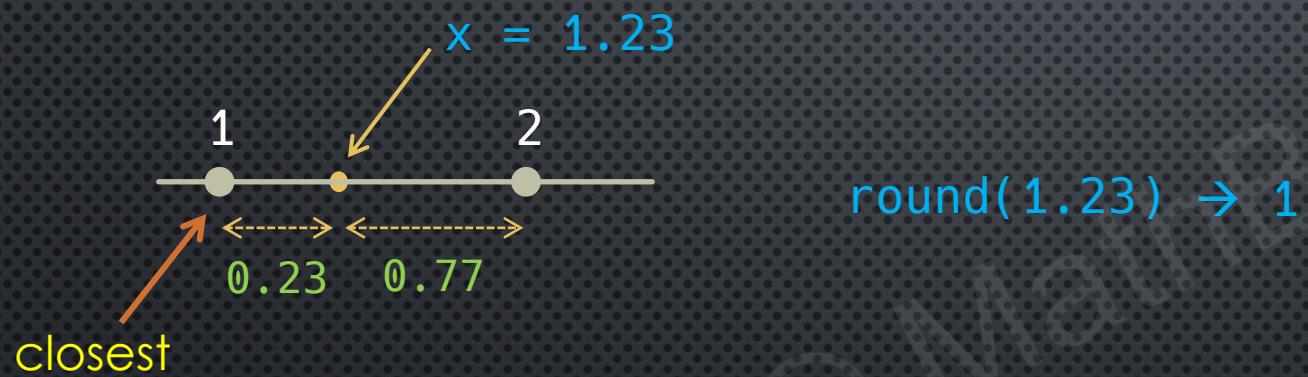
`round(x) → int`

`round(x, n) → same type as x`

`round(x, 0) → same type as x`

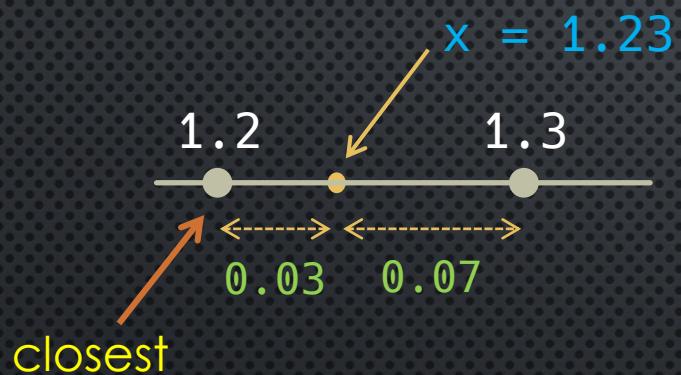
$n = 0$

round to the closest multiple of $10^{-0} = 1$



$n > 0$

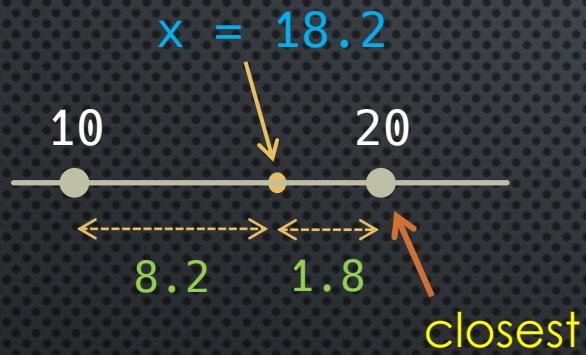
$n = 1$ round to the closest multiple of $10^{-1} = 0.1$



`round(1.23, 1) → 1.2`

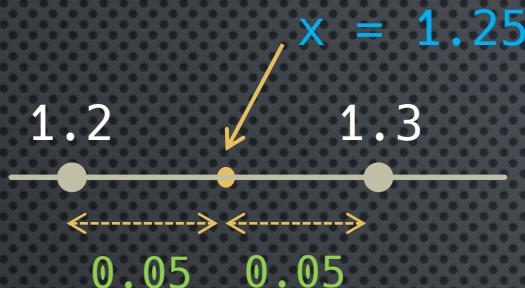
$n < 0$

$n = -1$ round to the closest multiple of $10^{-(-1)} = 10$



`round(18.2, -1) → 20`

Ties



`round(1.25, 1) = ???`

there is no closest value!!

We probably would expect `round(1.25, 1)` to be 1.3

rounding up / away from zero

Similarly, we would expect `round(-1.25, 1)` to result in -1.3

rounding down / away from zero

This type of rounding is called rounding to nearest, with ties away from zero

But in fact: `round(1.25, 1) → 1.2` towards 0

`round(1.35, 1) → 1.4` away from 0

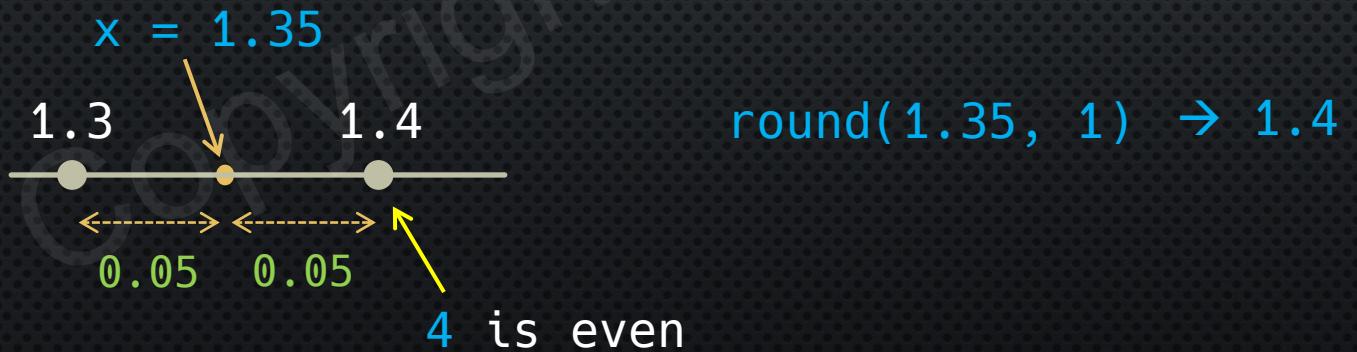
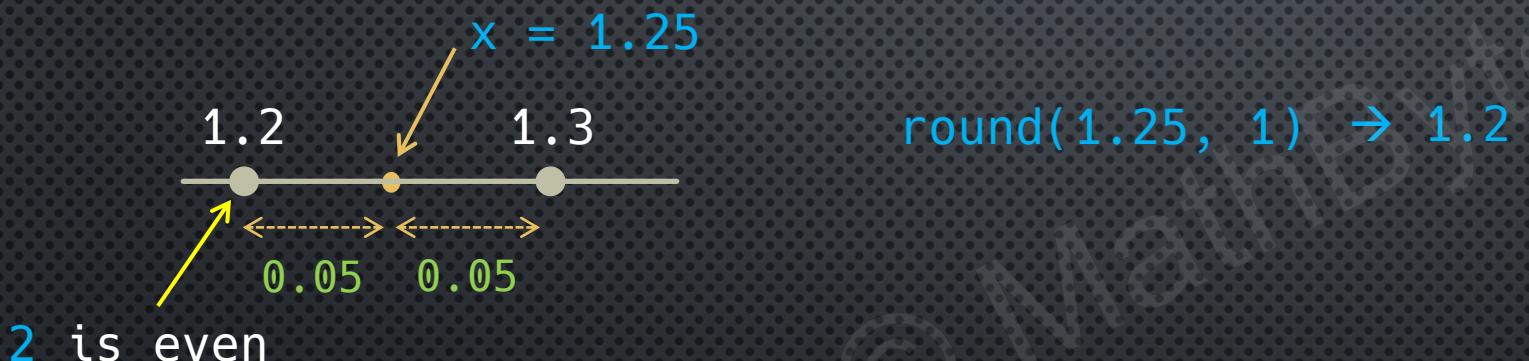
`round(-1.25, 1) → -1.2` towards 0

`round(-1.35, 1) → -1.4` away from 0

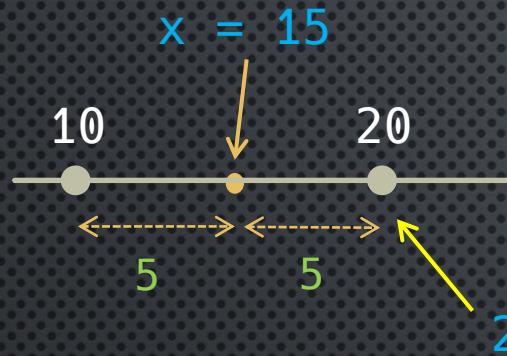


Banker's Rounding

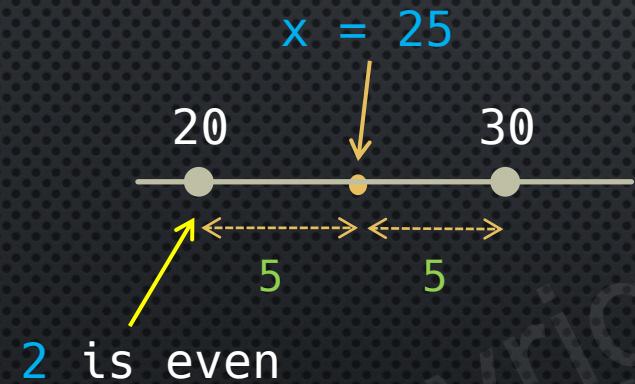
IEEE 754 standard: rounds to the nearest value, with ties rounded to the nearest value with an even least significant digit



$n = -1$ round to the closest multiple of $10^{-(-1)} = 10$



$\text{round}(15, -1) \rightarrow 20$



$\text{round}(25, -1) \rightarrow 20$

Why Banker's Rounding?

Less biased rounding than ties away from zero

Consider averaging three numbers, and averaging the rounded value of each:

$$0.5, 1.5, 2.5 \rightarrow \text{avg} = 4.5 / 3 = 1.5$$

"standard" rounding: 1, 2, 3 $\rightarrow \text{avg} = 6 / 3 = 2$

banker's rounding: 0, 2, 2 $\rightarrow \text{avg} = 4 / 3 = 1.3\dots$

If you really insist on rounding away from zero...

One common (and partially incorrect) way to round to nearest unit that often comes up on the web is:

$$\text{int}(x + 0.5) \quad 10.3 \rightarrow \text{int}(10.3 + 0.5) = \text{int}(10.8) = 10$$

$$10.9 \rightarrow \text{int}(10.9 + 0.5) = \text{int}(11.4) = 11$$

$$10.5 \rightarrow \text{int}(10.5 + 0.5) = \text{int}(11.0) = 11$$

but, this does not work for negative numbers

$$-10.3 \rightarrow \text{int}(-10.3 + 0.5) = \text{int}(-9.8) = -9$$

$$-10.9 \rightarrow \text{int}(-10.9 + 0.5) = \text{int}(-10.4) = -10$$

$$-10.5 \rightarrow \text{int}(-10.5 + 0.5) = \text{int}(-10.0) = -10$$

Technically, this is also an acceptable rounding method referred to as rounding towards + infinity

But this not rounding towards zero !!

If you really insist on rounding away from zero...

The correct way to do it:

`sign(x) * int(abs(x)+0.5)`

$$\text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

!! Not the same as the mathematical `sgn` (signum) function!

	10.4	10.5	10.6	-10.4	-10.5	-10.6
sign(x)	+	+	+	-	-	-
abs(x)+0.5	10.9	11.0	11.1	10.9	11.0	11.5
int(abs(x)+0.5)	10	11	11	10	11	11
sign(x) * int(abs(x)+0.5)	10	11	11	-10	-11	-11
= int(x + 0.5 * sign(x))						

Python does not have a `sign` function !

We can however use the `math.copysign()` function to achieve our goal:

`copysign(x, y)` returns the magnitude (absolute value) of `x` but with the sign of `y`

`sign(x) = copysign(1, x)`

```
sign(x) * int(abs(x)+0.5)

def round_up(x):
    from math import fabs, copysign
    return copysign(1, x) * int(fabs(x) + 0.5)
```

A simpler way to code this:

```
int(x + 0.5 * sign(x))

def round_up(x):
    from math import copysign
    return int(x + copysign(0.5, x))
```

DECIMALS

Copyright © McGraw-Hill Education

The decimal module

(PEP 327)

`float 0.1` → infinite binary expansion

$$(0.1)_{10} = (0.0\ 0011\ 0011\ 0011\ \dots)_2$$
$$= \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \dots$$

→ finite decimal expansion

$$(0.1)_{10} = \frac{1}{10}$$

alternative to using the (binary) `float` type → avoids the approximation issues with floats

finite number of significant digits → rational number (see videos on rationals)

So why not just use the `Fraction` class?

to add two fractions → common denominator

→ complex, requires extra memory

Why do we even care?

Why not just use binary floats?

finance, banking, and any other field where exact finite representations are highly desirable

let's say we are adding up all the financial transactions that took place over a certain time period

amount = \$100.01 1,000,000,000 transactions NYSE: 2-6 billion shares traded **daily**

100.01 → 100.0100000000000051159076975

sum → \$100010000000.00 (exact decimal)

\$100009998761.14639282226562500000000000 (approximate binary float)

\$1238.85... off!!

Decimals have a **context** that controls certain aspects of working with decimals

precision during arithmetic operations

rounding algorithm

This context can be **global** → the **default** context

or temporary (**local**) → sets temporary settings without affecting the global settings

```
import decimal
```

default context → `decimal.getcontext()`

local context → `decimal.localcontext(ctx=None)`

creates a new context, copied from ctx
or from default if ctx not specified

returns a context manager (use a **with** statement)

Precision and Rounding

`ctx = decimal.getcontext()` → context (global in this case)

`ctx.prec` → get or set the precision (value is an int)

`ctx.rounding` → get or set the rounding mechanism (value is a string)

ROUND_UP	rounds away from zero
ROUND_DOWN	rounds towards zero
ROUND_CEILING	rounds to ceiling (towards $+\infty$)
ROUND_FLOOR	rounds to floor (towards $-\infty$)
→ ROUND_HALF_UP	rounds to nearest, ties away from zero
→ ROUND_HALF_DOWN	rounds to nearest, ties towards zero
float rounding algorithm	rounds to nearest, ties to even (least significant digit)

Working with Global and Local Contexts

Global

```
decimal.getcontext().rounding = decimal.ROUND_HALF_UP  
//decimal operations performed here will use the current default context
```

Local

```
with decimal.localcontext() as ctx:  
    ctx.prec = 2  
    ctx.rounding = decimal.ROUND_HALF_UP  
  
//decimal operations performed here  
//will use the ctx context
```

DECIMALS

CONSTRUCTORS AND CONTEXTS

Constructing Decimal Objects

The `Decimal` class is in the `decimal` module

```
import decimal  
from decimal import Decimal
```

`Decimal(x)` `x` can be a variety of types

integers	<code>a = Decimal(10)</code>	→ 10
other Decimal object		
strings	<code>a = Decimal('0.1')</code>	→ 0.1
tuples	<code>a = Decimal((1, (3, 1, 4, 1, 5), -4))</code>	→ -3.1415
floats?	yes, but not usually done	

`Decimal(0.1)` → 0.10000000000000005551

Since 0.1 does not have an exact binary float representation it cannot be used to create an exact Decimal representation of itself

→ Use strings or tuples instead

Using the tuple constructor

1.23 → +123 × 10⁻²

-1.23 → -123 × 10⁻²

sign digits exponent

(s, (d₁, d₂, d₃, ...), exp)

$$s(x) = \begin{cases} 0 & \text{if } x \geq 0 \\ 1 & \text{if } x < 0 \end{cases}$$

Example: -3.1415 → (1, (3, 1, 4, 1, 5), -4)

a = Decimal((1, (3, 1, 4, 1, 5), -4)) a → -3.1415

Context Precision and the Constructor

Context precision affects mathematical operations

Context precision does not affect the constructor

```
import decimal
from decimal import Decimal

decimal.getcontext().prec = 2      ← global (default) context now has precision set to 2

a = Decimal('0.12345')    a → 0.12345
b = Decimal('0.12345')    b → 0.12345
c = a + b                a + b = 0.2469    c → 0.25
```

Local vs Global Context

```
import decimal
from decimal import Decimal

decimal.getcontext().prec = 6

a = Decimal('0.12345')
b = Decimal('0.12345')
print(a + b)                  → 0.24690

with decimal.localcontext() as ctx:
    ctx.prec = 2
    c = a + b
    print(c)                  → 0.25

print(c)                      → 0.25
```

DECIMALS

MATHEMATICAL OPERATIONS

Some arithmetic operators don't work the same as floats or integers

// and % → also `divmod()`

The // and % operators still satisfy the usual equation: $n = d * (n // d) + (n \% d)$

But for integers, the // operator performs floor division → `a // b = floor(a/b)`

For Decimals however, it performs truncated division → `a // b = trunc(a/b)`

negative
numbers!

`10 // 3 → 3` `Decimal(10) // Decimal(3) → Decimal(3)`

`-10 // 3 → -4` `Decimal(-10) // Decimal(3) → Decimal(-3)`

Boils down to the algorithm used to actually perform integer division

$$\begin{array}{r} a \\ \hline b \end{array} \quad \begin{array}{l} \text{dividend} \\ \text{divisor} \end{array}$$

- figure out the sign of the result
- use absolute values for divisor and dividend
- keep subtracting b from a as long as $a \geq b$
- return the signed number of times this was performed

$$\frac{10}{3} \quad \text{res is +} \quad \begin{array}{r} 10 - 3 \\ = 7 \end{array} \quad \begin{array}{r} 7 - 3 \\ = 4 \end{array} \quad \begin{array}{r} 4 - 3 \\ = 1 \end{array} \quad 1 < 3 - \text{STOP} \quad \text{return 3}$$

$$\frac{-10}{3} \quad \text{res is -} \quad \begin{array}{r} 10 - 3 \\ = 7 \end{array} \quad \begin{array}{r} 7 - 3 \\ = 4 \end{array} \quad \begin{array}{r} 4 - 3 \\ = 1 \end{array} \quad 1 < 3 - \text{STOP} \quad \text{return -3}$$

this is basically the same as truncating the real division

$$\text{trunc}(10/3) \rightarrow 3 \quad \text{trunc}(-10/3) \rightarrow -3$$

But $n = d * (n // d) + (n \% d)$ is still satisfied

$n = -135, d = 4$

	Integer	Decimal
$-135 // 4$	-34	-33
$-135 \% 4$	1	-3
$d * (n // d) + (n \% d)$	$4 * (-34) + 1 = -135$	$4 * (-33) + (-3) = -135$

Other Mathematical Operations

The Decimal class defines a bunch of various mathematical operations, such as sqrt, logs, etc.

But not all functions defined in the math module are defined in the Decimal class

E.g. trig functions

We can use the math module,

but Decimal objects will first be cast to floats

– so we lose the whole precision mechanism that made us use Decimal objects in the first place!

Usually will want to use the math functions defined in the Decimal class if they are available

```
decimal.getcontext().prec = 28  
  
x = 0.01  
x_dec = Decimal('0.01')  
  
root = math.sqrt(x)  
root_mixed = math.sqrt(x_dec)  
root_dec = x_dec.sqrt()  
  
print(format(root, '1.27f')) → 0.10000000000000005551115123  
print(format(root_mixed, '1.27f')) → 0.10000000000000005551115123  
print(root_dec) → 0.1  
  
print(format(root * root, '.27f')) → 0.01000000000000001942890293  
print(format(root_mixed * root_mixed, '.27f')) → 0.01000000000000001942890293  
print(root_dec * root_dec) → 0.01
```

DECIMALS

EASE AND PERFORMANCE CONSIDERATIONS

There are some drawbacks to the `Decimal` class vs the `float` class

- not as easy to code: construction via strings or tuples
- not all mathematical functions that exist in the math module have a Decimal counterpart
- more `memory` overhead
- `performance`: much slower than floats (relatively)

COMPLEX NUMBERS

Copyright © 2014

The **complex** class

```
Example:      a = complex(1, 2)
                  b = 1 + 2j
                  a == b → True
```

`x` and `y` (the real and imaginary parts) are stored as floats

Some instance properties and methods

.real → returns the real part

.imag → returns the imaginary part

.conjugate() → returns the complex conjugate

```
d = 2 - 3j
```

```
d.real → 2
```

```
d.imag → -3
```

```
d.conjugate() → 2 + 3j
```

Arithmetic Operators

The standard arithmetic operators (+, -, /, *, **) work as expected with complex numbers

$$(1 + 2j) + (3 + 4j) \rightarrow 4 + 6j$$

$$(1 + 2j) * (3 + 4j) \rightarrow 5 + 10j$$

Real and Complex numbers can be mixed:

$$(1 + 2j) + 3 \rightarrow 4 + 2j$$

$$(1 + 2j) * 3 \rightarrow 3 + 6j$$

// and % operators are not supported

Other operations

The `==` and `!=` operators are supported

Comparison operators such as `<`, `>`, `<=` and `>=` are **not** supported

Functions in the `math` module will **not** work

Use the `cmath` module instead

exponentials

logs

trigs and inverse trigs

hyperbolics and inverse hyperbolics

polar / rectangular conversions

`isclose`

Rectangular to Polar

```
import cmath
```

`cmath.phase(x)` Returns the argument (phase) φ of the complex number x

$\varphi \in [-\pi, \pi]$ measured counter-clockwise from the real axis

`abs(x)` Returns the magnitude (r) of x

```
a = -1 + 0j
```

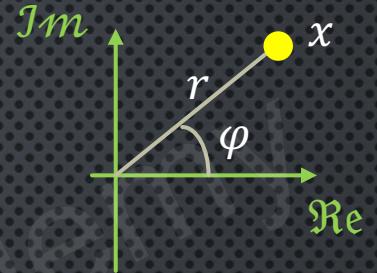
```
cmath.phase(a) → 3.1415... (π)    abs(a) → 1
```

```
a = -1j
```

```
cmath.phase(a) → -1.570... (- $\frac{\pi}{2}$ )    abs(a) → 1
```

```
a = 1 + 1j
```

```
cmath.phase(a) → 0.785... ( $\frac{\pi}{4}$ )    abs(a) → 1.414... ( $\sqrt{2}$ )
```



Polar to Rectangular

```
import cmath
```

`cmath.rect(r, phi)`

Returns a complex number (rectangular coordinates) equivalent to the complex number defined by (r, phi) in polar coordinates

```
cmath.rect(math.sqrt(2), math.pi/4) → 1 + 1j
```

`1.0000000000000002+1.0000000000000002j`

Euler's Identity

$$e^{i\pi} + 1 = 0$$

```
cmath.exp(cmath.pi * 1j) + 1
```

```
→ 1.2246467991473532e-16j binary floats tend to spoil the effect!
```

So, the next best thing:

```
cmath.isclose(cmath.exp(cmath.pi*1j) + 1, 0, abs_tol=0.0001) → True
```

Do note however the same issue with `isclose()` as we discussed in the float videos:

```
cmath.isclose(cmath.exp(cmath.pi*1j) + 1, 0) → False
```

BOOLEANS

INTEGER SUBCLASS

Python has a concrete `bool` class that is used to represent Boolean values.

However, the `bool` class is a `subclass` of the `int` class

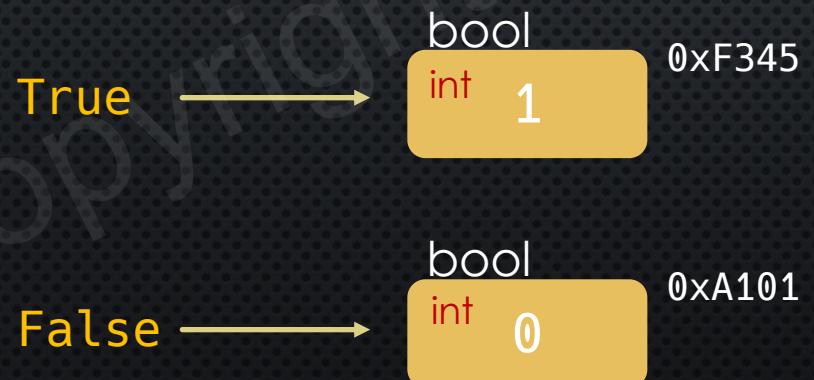
```
issubclass(bool, int) → True
```

Two constants are defined: `True` and `False`

They are singleton objects of type `bool`

i.e. they posses all the properties and methods of integers, and add some specialized ones such as `and`, `or`, etc

```
isinstance(True, bool) → True  
isinstance(True, int) → True
```



`is` vs `==`

Because `True` and `False` are singleton objects, they will always retain their same memory address throughout the lifetime of your application

So, comparisons of any Boolean expression to `True` and `False` can be performed using either the `is` (identity) operator, or the `==` (equality) operator

`a == True` `a is True` where `a` is a `bool` object

But since `bool` objects are also `int` objects, they can also be interpreted as the integers `1` and `0`

`int(True) → 1` `int(False) → 0`

But: `True` and `1` are not the same objects

`id(True) ≠ id(1)`

`False` and `0` are not the same objects

`id(False) ≠ id(0)`

`True is 1 → False`

`True == 1 → True`

Booleans as Integers

This can lead to "strange" behavior you may not expect!

`True > False → True`

`(1 == 2) == False → True` `(1 == 2) == 0 → True`

In fact, any integer operation will also work with booleans (`//`, `%`, etc)

`True + True + True → 3` `(True + True + True) % 2 → 1`

`-True → -1`

`100 * False → 0`

The Boolean constructor

The Boolean constructor `bool(x)` returns `True` when `x` is `True`, and `False` when `x` is `False`

Wow, that sounds like a useless constructor! But not at all!

What really happens is that many classes contain a definition of how to cast instances of themselves to a Boolean – this is sometimes called the `truth value` (or `truthiness`) of an object
(upcoming video)

Integers have a truth value defined according to this rule:

`bool(0) → False` (`0` is falsy)

`bool(x) → True` for any `int x ≠ 0` (`x` is truthy)

Examples

`bool(0) → False`

`bool(1) → True`

`bool(100) → True`

`bool(-1) → True`

BOOLEANS

© OBJECT TRUTH VALUES

Objects have Truth Values

All objects in Python have an associated truth value

We already saw this with integers (although to be fair, `bool` is a subclass of `int`)

But this works the same for any object

In general, the rules are straightforward

Every object has a `True` truth value, except:

- `None`
- `False`
- `0` in any numeric type (e.g. `0`, `0.0`, `0+0j`, ...)
- empty sequences (e.g. `list`, `tuple`, `string`, ...)
- empty mapping types (e.g. `dictionary`, `set`, ...)
- custom classes that implement a `__bool__` or `__len__` method that returns `False` or `0`

which have a `False` truth value

Under the hood

Classes define their truth values by defining a special instance method:

`__bool__(self)` (or `__len__`)

Then, when we call `bool(x)` Python will actually executes `x.__bool__()`

or `__len__` if `__bool__` is not defined

if neither is defined, then `True`

Example: Integers

```
def __bool__(self):  
    return self != 0
```

When we call `bool(100)` Python actually executes `100.__bool__()`

and therefore returns the result of `100 != 0` which is `True`

When we call `bool(0)` Python actually executes `0.__bool__()`

and therefore returns the result of `0 != 0` which is `False`

We will cover this and many other special functions in a later section

Examples

`bool([1, 2, 3]) → True`

`bool([]) → False`

`bool(None) → False`

`bool('abc') → True`

`bool('') → False`

`bool(0) → False`

`bool(0 + 0j) → False`

`bool(Decimal('0.0')) → False`

`bool(-1) → True`

`bool(1 + 2j) → True`

`bool(Decimal('0.1')) → True`

`if my_list:
 # code block`

code block will execute if and only if `my_list` is both not `None` and not empty

this is equivalent to:

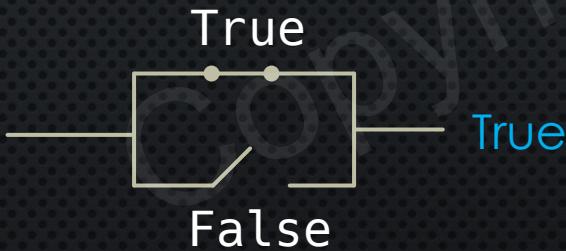
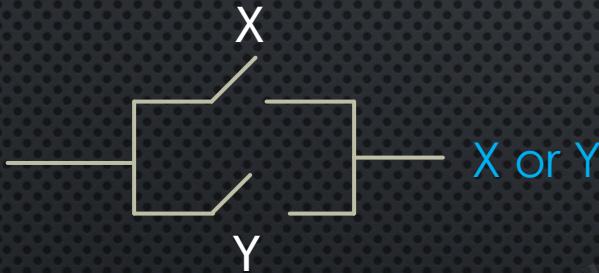
`if my_list is not None and len(my_list) > 0:
 # code block`

BOOLEANS

OPERATORS, PRECEDENCE AND SHORT CIRCUIT EVALUATION

The Boolean Operators: `not`, `and`, `or`

X	Y	not X	X and Y	X or Y
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1



Commutativity

$$A \text{ or } B == B \text{ or } A$$

$$A \text{ and } B == B \text{ and } A$$

Distributivity

$$A \text{ and } (B \text{ or } C) == (A \text{ and } B) \text{ or } (A \text{ and } C)$$

$$A \text{ or } (B \text{ and } C) == (A \text{ or } B) \text{ and } (A \text{ or } C)$$

Associativity

$$A \text{ or } (B \text{ or } C) == (A \text{ or } B) \text{ or } C$$

$$A \text{ and } (B \text{ and } C) == (A \text{ and } B) \text{ and } C$$

$$A \text{ or } B \text{ or } C \rightarrow (A \text{ or } B) \text{ or } C$$

$$A \text{ and } B \text{ and } C \rightarrow (A \text{ and } B) \text{ and } C$$

left-to-right evaluation

De Morgan's Theorem

$$\text{not}(A \text{ or } B) == (\text{not } A) \text{ and } (\text{not } B)$$

$$\text{not}(A \text{ and } B) == (\text{not } A) \text{ or } (\text{not } B)$$

Miscellaneous

$$\text{not}(x < y) == x \geq y \quad \text{not}(x \leq y) == x > y$$

$$\text{not}(x > y) == x \leq y \quad \text{not}(x \geq y) == x < y$$

$$\text{not}(\text{not } A) == A$$

Operator Precedence

()

< > <= >= == != in is

not

and

or

highest
precedence
lowest

True or True and False

→ True or False → True

(True or True) and False

→ True and False → False

When in doubt, or to be absolutely sure, use parentheses!

Also, use parentheses to make your code more human readable!

a < b or a > c and not x or y



(a < b) or ((a > c) and (not x)) or y



True or (True and False)

Short-Circuit Evaluation

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1



if **X** is **True**, then **X or Y** will be **True** no matter the value of **Y**
So, **X or Y** will return **True** without evaluating **Y** if **X** is **True**

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1



if **X** is **False**, then **X and Y** will be **False** no matter the value of **Y**
So, **X and Y** will return **False** without evaluating **Y** if **X** is **False**

Example

Scenario: There is some data feed that lists a stock symbol, and some financial data.

Your job is to monitor this feed, looking for specific stock symbols defined in some watch list, and react only if the current stock price is above some threshold. Getting the current stock price has an associated cost.

If Boolean expressions did not implement short-circuiting, you would probably write:

```
if symbol in watch_list:  
    if price(symbol) > threshold:  
        # do something
```

since calling the `price()` method has a cost,
you would only want to call it if the symbol was
on your watch list

But because of short-circuit evaluation you could write this equivalently as:

```
if symbol in watch_list and price(symbol) > threshold:  
    # do something
```

Example

`name` is a string returned from a nullable text field in a database

perform some action if the first character of name is a digit (0-9)

`null` → `None`

`,`

`'abc'`

```
if name[0] in string.digits:  
    # do something
```

this code will break if `name` is `None` or an empty string

because of short-circuiting and truth values

```
if name and name[0] in string.digits:  
    # do something
```

if `name` is falsy (either `None` or an empty string) then
`name[0] in string.digits` is not evaluated

BOOLEANS

BOOLEAN OPERATORS IN PYTHON

Boolean Operators and Truth Values

X	Y	X and Y	X or Y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Normally, Boolean operators are defined to operate on and return Boolean values

True or False → True

a = 2 a > 0 and b < 5 → True
b = 3

But every object in Python has a truth value (truthiness)

so, for any object X and Y, we could also write

bool(X) and bool(Y) bool(X) or bool(Y)

In fact, we don't need to use bool()

X and Y X or Y

So, what is returned when evaluating these expressions?

A Boolean? No!

Definition of `or` in Python

`X or Y`

If `X` is **truthy**, returns `X`, otherwise returns `Y`



Does this work as expected when X and Y are Boolean values?

X	Y	X or Y
0	0	0
0	1	1
1	0	1
1	1	1

X	Y	Rule	Result
0	0	X is False, so return Y	0
0	1	X is False, so return Y	1
1	0	X is True, so return X	1
1	1	X is True, so return X	1



If `X` is **truthy**, returns `X`, otherwise evaluates `Y` and returns it

Definition of `and` in Python

`X and Y` If `X` is falsy, returns `X`, otherwise returns `Y`

X	Y	X and Y
0	0	0
0	1	0
1	0	0
1	1	1

Does this work as expected when X and Y are Boolean values?

X	Y	Rule	Result
0	0	X is False, so return X	0
0	1	X is False, so return X	0
1	0	X is True, so return Y	0
1	1	X is True, so return Y	1



If `X` is falsy, returns `X`, otherwise evaluates `Y` and returns it

Consequence: or

X or Y If X is truthy, returns X, otherwise evaluates and returns Y

X	Y	X or Y
None	'N/A'	'N/A'
''	'N/A'	'N/A'
'hello'	'N/A'	'hello'

a = s or 'N/A'

if s is None

a → N/A

if s is ''

a → N/A

if s is a string
with characters

a → s

i.e. a will either be s or 'N/A' if s is None or an empty string

Example

We can expand this further:

```
a = s1 or s2 or s3 or 'N/A'
```

In this case, **a** will be equal to the first truthy value (left to right evaluation)

and is guaranteed to have a value, since '**N/A**' is truthy

Example

We have an integer variable **a** that cannot be zero – if it is zero, we want to set it to 1.

```
a = a or 1
```

Consequence: and

X and Y If X is falsy, returns X, otherwise evaluates and returns Y

X	Y	X and Y
10	$20/X$	2
0	$20/X$	0

Seems like we are able to avoid a division by zero error using the **and** operator

`x = a and total/a`

`a = 10 → x = 10 and total/10 → total/10`

`a = 0 → x = 0 and total/0 → 0`

Example

Computing an average

`sum, n` Sometimes `n` is non-zero, sometimes it is

In either case: `avg = n and sum/n`

Example

You want to return the first character of a string `s`, or an empty string if the string is `None` or empty

Option 1

```
if s:  
    return s[0]  
else:  
    return ''
```

Option 2

```
return s and s[0] → doesn't handle None case  
return (s and s[0]) or ''
```



The Boolean `not`

`not` is a built-in function that returns a Boolean value

`not x` → `True` if `x` is falsy

→ `False` if `x` is truthy

`[]` → falsy

`not []` → `True`

`[1, 2]` → truthy

`not [1, 2]` → `False`

`None` → falsy

`not None` → `True`

COMPARISON OPERATORS

Copyright © 2014

Categories of Comparison Operators

- binary operators
- evaluate to a `bool` value

Identity Operations

`is` `is not`

compares memory address – any type

Value Comparisons

`==` `!=`

compares values – different types OK,
but must be compatible

Ordering Comparisons

`<` `<=` `>` `>=`

doesn't work for all types

Membership Operations

`in` `not in`

used with iterable types

Numeric Types

We will examine other types, including iterables, later in this course

Value comparisons will work with all numeric types

Mixed types (except complex) in value and ordering comparisons is supported

Note: Value equality operators work between floats and Decimals, but as we have seen before, using value equality with floats has some issues!

```
10.0 == Decimal('10.0') → True
```

 0.1 == Decimal('0.1') → False

```
Decimal('0.125') == Fraction(1, 8) → True
```

```
True == 1 → True
```

```
True == Fraction(3, 3) → True
```

Ordering Comparisons

Again, these work across all numeric types, except for complex numbers

```
1 < 3.14 → True
```

```
Fraction(22, 7) > math.pi → True
```

```
Decimal('0.5') <= Fraction(2, 3) → True
```

```
True < Decimal('3.14') → True
```

```
Fraction(2, 3) > False → True
```

Chained Comparisons

`a == b == c → a == b and b == c`

`a < b < c → a < b and b < c`

`1 == Decimal('1.0') == Fraction(1,1) → True`

`1 == Decimal('1.5') == Fraction(3, 2) → False`

`1 < 2 < 3 → 1 < 2 and 2 < 3 → True`

`1 < math.pi < Fraction(22, 7)`

`→ 1 < math.pi and math.pi < Fraction(22, 7)`

`→ True`

Chained Comparisons

$a < b > c \rightarrow a < b \text{ and } b > c$

$5 < 6 > 2 \rightarrow 5 < 6 \text{ and } 6 > 2 \rightarrow \text{True}$

$5 < 6 > 10 \rightarrow 5 < 6 \text{ and } 6 > 10 \rightarrow \text{False}$

$a < b < c < d \rightarrow a < b \text{ and } b < c \text{ and } c < d$

$1 < 2 < 3 < 4 \rightarrow 1 < 2 \text{ and } 2 < 3 \text{ and } 3 < 4 \rightarrow \text{True}$

$1 < 10 > 4 < 5 \rightarrow 1 < 10 \text{ and } 10 > 4 \text{ and } 4 < 5 \rightarrow \text{True}$

```
if my_min == cnt < val > other <= my_max not in lst:  
    # do something
```



FUNCTION PARAMETERS

Copyright © 2014

Arguments vs Parameters

Positional vs Keyword-Only Arguments

Optional Arguments via Defaults

Unpacking Iterables and Function Arguments

Extended Unpacking

Variable Number of Positional and Keyword-Only Arguments

ARGUMENT vs PARAMETER

Semantics!

```
def my_func(a, b):  
    # code here
```

In this context, **a** and **b** are called **parameters** of **my_func**

Also note that **a** and **b** are **variables**, local to **my_func**

When we call the function:

```
x = 10  
y = 'a'  
  
my_func(x, y)
```

x and **y** are called the **arguments** of **my_func**

Also note that **x** and **y** are passed by **reference**

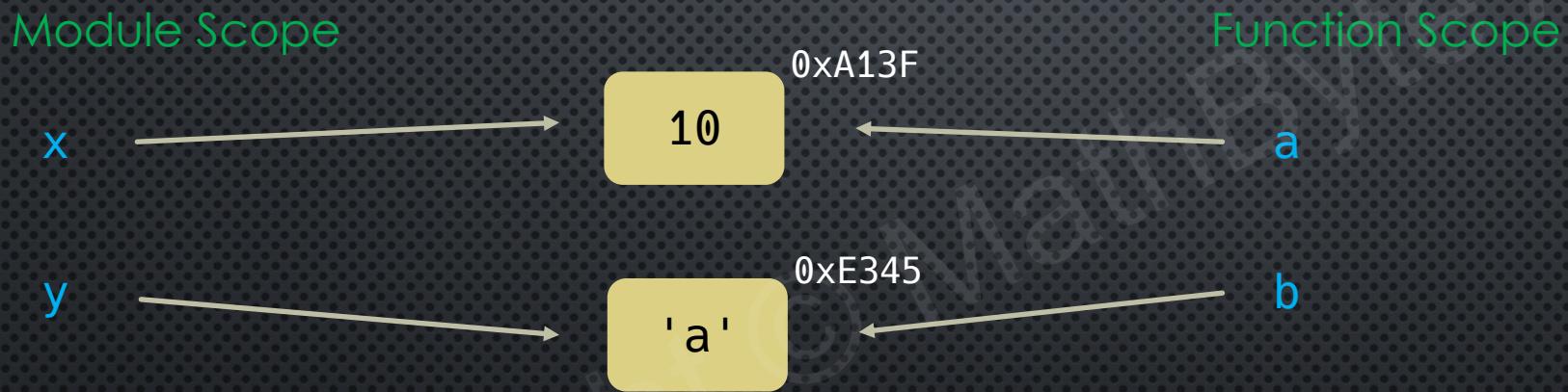
i.e. the **memory addresses** of **x** and **y** are passed

It's OK if you mix up these terms – everyone will understand what you mean!

```
x = 10  
y = 'a'
```

```
my_func(x, y)
```

```
def my_func(a, b):  
    # code here
```



POSITIONAL AND KEYWORD ARGUMENTS

Copyright © 2014, Packt Publishing Ltd.

Positional Arguments

Most common way of assigning arguments to parameters: via the **order** in which they are passed
i.e. their **position**

```
def my_func(a, b):  
    # code ...
```

my_func(10, 20) → a = 10, b = 20

my_func(20, 10) → a = 20, b = 10

Default Values

A positional arguments can be made optional by specifying a default value for the corresponding parameter

```
def my_func(a, b=100):      my_func(10, 20)    → a = 10, b = 20  
    # code ...  
                          my_func(5)        → a = 5, b = 100
```

Consider a case where we have three arguments, and we want to make one of them optional:

```
def my_f c(a, b=100, c):  
    # cod ..
```

How would we call this function without specifying a value for the second parameter?

```
my_fu (5, 5) ???
```

If a positional parameter is defined with a default value

every positional parameter after it

must also be given a default value

Default Values

```
def my_func(a, b=5, c=10):  
    # code ...  
  
    my_func(1)          → a = 1, b = 5, c = 10  
    my_func(1, 2)        → a = 1, b = 2, c = 10  
  
    my_func(1, 2, 3)    → a = 1, b = 2, c = 3
```

But what if we want to specify the 1st and 3rd arguments, but omit the 2nd argument?

i.e. we want to specify values for **a** and **c**, but let **b** take on its default value?

→ Keyword Arguments (named arguments)

my_func(a=1, c=2) → a = 1, b = 5, c = 2

my_func(1, c=2) → a = 1, b = 5, c = 2

Keyword Arguments

Positional arguments can, **optionally**, be specified by using the parameter name
whether or not the parameters have default values

```
def my_func(a, b, c)      my_func(1, 2, 3)  
                          my_func(1, 2, c=3)           → a=1, b=2, c=3  
                          my_func(a=1, b=2, c=3)  
                          my_func(c=3, a=1, b=2)
```

But once you use a named argument, all arguments **thereafter must** be named too

my_func(c=1, 2, 3)	✗
my_func(1, b=2, 3)	✗
my_func(1, b=2, c=3)	✓
my_func(1, c=3, b=2)	✓

Keyword Arguments

All arguments after the first named (keyword) argument, must be named too

Default arguments may still be omitted

```
def my_func(a, b=2, c=3)
```

my_func(1) → a=1, b=2, c=3

my_func(a=1, b=5) → a=1, b=5, c=3

my_func(c=0, a=1) → a=1, b=2, c=0

UNPACKING ITERABLES

Copyright © 2018

A Side Note on Tuples

(1, 2, 3)

What defines a tuple in Python, is not (), but ,

1, 2, 3 is also a tuple → (1, 2, 3) The () are used to make the tuple clearer

To create a tuple with a single element:

(1) will not work as intended → int

1, or (1,) → tuple

The only exception is when creating an empty tuple: () or tuple()

Packed Values

Packed values refers to values that are **bundled** together in some way

Tuples and Lists are obvious

```
t = (1, 2, 3)
```

```
l = [1, 2, 3]
```

Even a string is considered to be a packed value:

```
s = 'python'
```

Sets and dictionaries are also packed values:

```
set1 = {1, 2, 3}
```

```
d = {'a': 1, 'b': 2, 'c': 3}
```

In fact, any **iterable** can be considered a packed value

Unpacking Packed Values

Unpacking is the act of **splitting** packed values into **individual variables** contained in a list or tuple

`a, b, c = [1, 2, 3]` 3 elements in `[1, 2, 3]` → need 3 variables to unpack



this is actually a tuple of 3 variables: `a`, `b` and `c`

`a → 1` `b → 2` `c → 3`

The unpacking into individual variables is based on the relative **positions** of each element

Does this remind you of how positional arguments were assigned to parameters in function calls?

Unpacking other Iterables

```
a, b, c = 10, 20, 'hello' → a = 10    b = 20    c = 'hello'
```

this is actually a tuple containing 3 values

```
a, b, c = 'XYZ' → a = 'X'    b = 'Y'    c = 'Z'
```

instead of writing a = 10 we can write a, b = 10, 20
 b = 20

In fact, unpacking works with any **iterable** type

```
for e in 10, 20, 'hello' → loop returns 10, 20, 'hello'
```

```
for e in 'XYZ' → loop returns 'X', 'Y', 'Z'
```

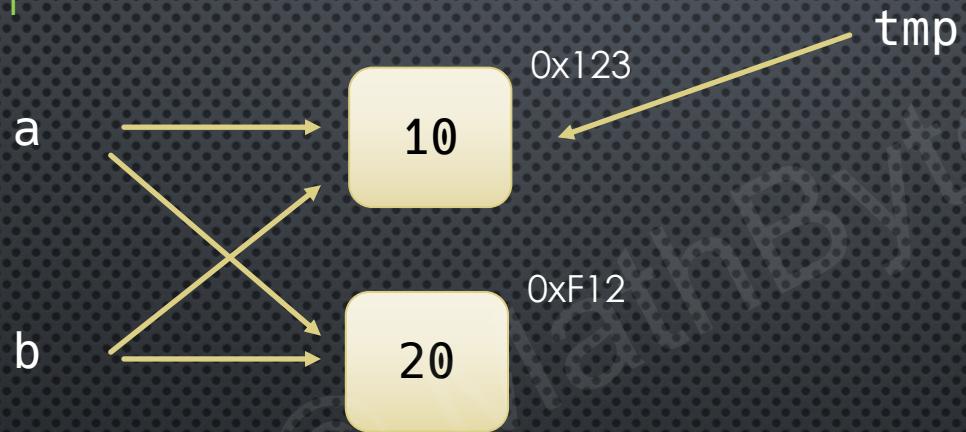
Simple Application of Unpacking

swapping values of two variables

a = 10
b = 20 → b = 20
a = 10

"traditional" approach

```
tmp = a  
a = b  
b = tmp
```



using unpacking

```
a, b = b, a
```



this works because in Python, the entire RHS is evaluated **first** and **completely**

then assignments are made to the LHS

Unpacking Sets and Dictionaries

```
d = {'key1': 1, 'key2': 2, 'key3': 3}
```

```
for e in d      → e iterates through the keys: 'key1', 'key2', 'key3'
```

so, when unpacking **d**, we are actually unpacking the **keys** of **d**

```
a, b, c = d      → a = 'key1', b = 'key2', c='key3'
```

or → a = 'key2', b = 'key1', c='key3'

or → a = 'key3', b = 'key1', c='key2'

etc...



Dictionaries (and Sets) are **unordered** types.

They can be iterated, but there is **no guarantee** the order of the results will match your literal!

In practice, we rarely unpack sets and dictionaries in precisely this way.

Example using Sets

```
s = {'p', 'y', 't', 'h', 'o', 'n'}
```

```
for c in s:           → p  
    print(c)          t  
                      h  
                      n  
                      o  
                      y
```

```
a, b, c, d, e, f = s      a = 'p'  
                           b = 't'  
                           c = 'h'  
                           ...  
                           f = 'y'
```

EXTENDED UNPACKING

USING THE `*` AND `**` OPERATORS

The use case for *

Much of this section applies to Python >= 3.5

We don't always want to unpack every single item in an iterable

We may, for example, want to unpack the first value, and then unpack the remaining values into another variable

```
l = [1, 2, 3, 4, 5, 6]
```

We can achieve this using slicing:

```
a = l[0]  
b = l[1:]
```

or, using simple unpacking:

```
a, b = l[0], l[1:]
```

(aka parallel assignment)

We can also use the * operator:

```
a, *b = l
```

Apart from cleaner syntax, it also works with any iterable, not just sequence types!

Usage with ordered types

a, *b = [-10, 5, 2, 100]

a, *b = (-10, 5, 2, 100)

a, *b = 'XYZ'

a = -10 b = [5, 2, 100]

a = -10 b = [5, 2, 100]

a = 'X' b = ['Y', 'Z']

this is still a list!

this is also a list!

The following also works:

a, b, *c = 1, 2, 3, 4, 5

a = 1 b = 2 c = [3, 4, 5]

a, b, *c, d = [1, 2, 3, 4, 5]

a = 1 b = 2 c = [3, 4] d = 5

a, *b, c, d = 'python'

a = 'p' b = ['y', 't', 'h']

c = 'o' d = 'n'

The `*` operator can only be used once in the LHS of an unpacking assignment

For obvious reason, you cannot write something like this:

```
a, *b, *c = [1, 2, 3, 4, 5, 6]
```

Since both `*b` and `*c` mean "the rest", both cannot exhaust the remaining elements

Usage with ordered types

We have seen how to use the `*` operator in the LHS of an assignment to unpack the RHS

```
a, *b, c = {1, 2, 3, 4, 5}
```

However, we can also use it this way:

```
l1 = [1, 2, 3]
l2 = [4, 5, 6]
l = [*l1, *l2]      → l = [1, 2, 3, 4, 5, 6]
```

```
l1 = [1, 2, 3]
l2 = 'XYZ'
l = [*l1, *l2]      → l = [1, 2, 3, 'X', 'Y', 'Z']
```

Usage with **unordered** types

Types such as sets and dictionaries have **no ordering**



```
s = {10, -99, 3, 'd'}  
print(s)           → {10, 3, 'd', -99}
```

Sets and dictionary keys are still iterable, but iterating has no guarantee of preserving the order in which the elements were created/added

But, the `*` operator still works, since it works with any iterable

```
s = {10, -99, 3, 'd'}  
a, *b, c = s      a = 10     b = [3, 'd']    c = -99
```

In practice, we rarely unpack sets and dictionaries directly in this way.

Usage with **unordered** types

It is useful though in a situation where you might want to create single collection containing all the items of multiple sets, or all the keys of multiple dictionaries

```
d1 = {'p': 1, 'y': 2}
```

```
d2 = {'t': 3, 'h': 4}
```

```
d3 = {'h': 5, 'o': 6, 'n': 7}
```

Note that the key 'h' is in both **d2** and **d3**

```
l = [*d1, *d2, *d3] → ['p', 'y', 't', 'h', 'h', 'o', 'n']
```

```
s = {*d1, *d2, *d3} → {'p', 'y', 't', 'h', 'o', 'n'} (order is not guaranteed)
```

The ****** unpacking operator

When working with dictionaries we saw that ***** essentially iterated the **keys**

```
d = {'p': 1, 'y': 2, 't': 3, 'h': 4}
```

```
a, *b = d
```

```
a = 'p'      b = ['y', 't', 'h']
```

(again, order is not guaranteed)

We might ask the question: can we unpack the **key-value** pairs of the dictionary?

Yes!

We need to use the ****** operator

Using **

```
d1 = {'p': 1, 'y': 2}
```

```
d2 = {'t': 3, 'h': 4}
```

```
d3 = {'h': 5, 'o': 6, 'n': 7}
```

Note that the key 'h' is in both d2 and d3

```
d = {**d1, **d2, **d3}    (note that the ** operator cannot be used in the LHS of an assignment)
```

```
→ {'p': 1, 'y': 2, 't': 3, 'h': 5, 'o': 6, 'n': 7}
```

Note that the value of 'h' in d3 "overwrote" the first value of 'h' found in d2

(order not guaranteed)

Using **

You can even use it to add key-value pairs from one (or more) dictionary into a dictionary literal:

```
d1 = {'a': 1, 'b': 2}
```

```
{'a': 10, 'c': 3, **d1}    → {'a': 1, 'b': 2, 'c': 3}
```

```
{**d1, 'a': 10, 'c': 3}    → {'a': 10, 'b': 2, 'c': 3}
```

(order not guaranteed)

Nested Unpacking

Python will support nested unpacking as well.

`l = [1, 2, [3, 4]]` Here, the third element of the list is itself a list.

We can certainly unpack it this way: `a, b, c = l` `a = 1` `b = 2` `c = [3, 4]`

We could then unpack `c` into `d` and `e` as follows: `d, e = c` `d = 3` `e = 4`

Or, we could simply do it this way: `a, b, (c, d) = [1, 2, [3, 4]]` `a = 1` `b = 2`
`c = 3` `d = 4`

Since strings are iterables too: `a, *b, (c, d, e) = [1, 2, 3, 'XYZ']`

`a = 1` `b = [2, 3]` `c, d, e = 'XYZ'`

→ `c = 'X'` `d = 'Y'` `e = 'Z'`

The `*` operator can only be used once in the LHS of an unpacking assignment

How about something like this then?

```
a, *b, (c, *d) = [1, 2, 3, 'python']
```

Although this looks like we are using `*` twice in the same expression, the second `*` is actually in a nested unpacking – so that's OK

```
a = 1      b = [2, 3]      c, *d = 'python'
```

```
→      c = 'p'  
          d = ['y', 't', 'h', 'o', 'n']
```

Try doing the same thing using slicing...

*args

Recall from iterable unpacking

```
a, b, c = (10, 20, 30) → a = 10    b = 20    c = 30
```

Something similar happens when **positional** arguments are passed to a function:

```
def func1(a, b, c):  
    # code
```

```
func1(10, 20, 30) → a = 10    b = 20    c = 30
```

*args

Recall also: `a, b, *c = 10, 20, 'a', 'b'` → `a=10` `b=20` `c=['a', 'b']`

Something similar happens when **positional** arguments are passed to a function:

```
def func1(a, b, *c):  
    # code
```

`func1(10, 20, 'a', 'b')` → `a=10` `b=20`

this is a **tuple**, not a list

`c=('a', 'b')`

The `*` parameter name is arbitrary – you can make it whatever you want

It is **customary** (but not required) to name it `*args`

```
def func1(a, b, *args):  
    # code
```

`*args` exhausts positional arguments

You cannot add more positional arguments after `*args`

```
def func1(a, b, *args, d):  
    # code
```

this is actually OK – covered in next lecture

This will not work!

```
func1(10, 20, 'a', 'b', 100)
```



Unpacking arguments

```
def func1(a, b, c):  
    # code
```

```
l = [10, 20, 30]
```

This will **not** work:

func1(l) 

But we can unpack the list **first** and **then** pass it to the function

func1(*l) → a = 10 b = 20 c = 30

KEYWORD ARGUMENTS

Copyright © 2018

Keyword Arguments

Recall that positional parameters can, optionally be passed as named (keyword) arguments

```
def func(a, b, c):  
    # code
```

func(1, 2, 3) → a = 1, b = 2, c = 3

func(a=1, c=3, b=2) → a = 1, b = 2, c = 3

Using named arguments in this case is entirely up to the caller.

Mandatory Keyword Arguments

We can make keyword arguments mandatory.

To do so, we create parameters after the positional parameters have been exhausted.

```
def func(a, b, *args, d):  
    #code
```

In this case, `*args` effectively exhausts all positional arguments

and `d` must be passed as a keyword (named) argument

```
func(1, 2, 'x', 'y', d = 100)  
→ a = 1, b = 2, args = ('x', 'y'), d = 100
```

```
func(1, 2, d = 100)  
→ a = 1, b = 2, args = (), d = 100
```

func(1, 2)  `d` was not a keyword argument

We can even omit any mandatory positional arguments:

```
def func(*args, d):  
    #code
```

func(1, 2, 3, d=100) → args = (1, 2, 3), d = 100

func(d=100) → args = (), d = 100

In fact we can force no positional arguments at all:

```
def func(*, d):  
    #code
```

* indicates the "end" of positional arguments

func(1, 2, 3, d=100) → Exception ✗

func(d=100) → d = 100 ✓

Putting it together

```
def func(a, b=1, *args, d, e=True):  
    # code
```

```
def func(a, b=1, *, d, e=True):  
    # code
```

a: mandatory positional argument (may be specified using a named argument)

b: optional positional argument (may be specified positionally, as a named argument, or not at all), defaults to **1**

args: catch-all for any (optional) additional positional arguments

*****: no additional positional arguments allowed

d: mandatory keyword argument

e: optional keyword argument, defaults to **True**

**kwargs

****kwargs**

***args** is used to scoop up variable amount of remaining positional arguments

→ tuple

The parameter name **args** is arbitrary – ***** is the real performer here

****kwargs** is used to scoop up a variable amount of remaining keyword arguments

→ dictionary

The parameter name **kwargs** is arbitrary – ****** is the real performer here

****kwargs** can be specified even if the positional arguments have **not** been exhausted

(unlike keyword-only arguments)

No parameters can come **after **kwargs**

Example

```
def func(*, d, **kwargs):  
    # code
```

```
func(d=1, a=2, b=3)      → d = 1  
                        kwargs = {'a': 2, 'b': 3}
```

```
func(d=1)                → d = 1  
                        kwargs = {}
```

Example

```
def func(**kwargs):  
    # code
```

```
func(a=1, b=2, c=3)      → kwargs = {'a': 1, 'b': 2, 'c': 3}  
func()                  → kwargs = {}
```

```
def func(*args, **kwargs):  
    # code
```

```
func(1, 2, a=10, b=20)  → args = (1, 2)  
                           kwargs = {'a': 10, 'b': 20}
```

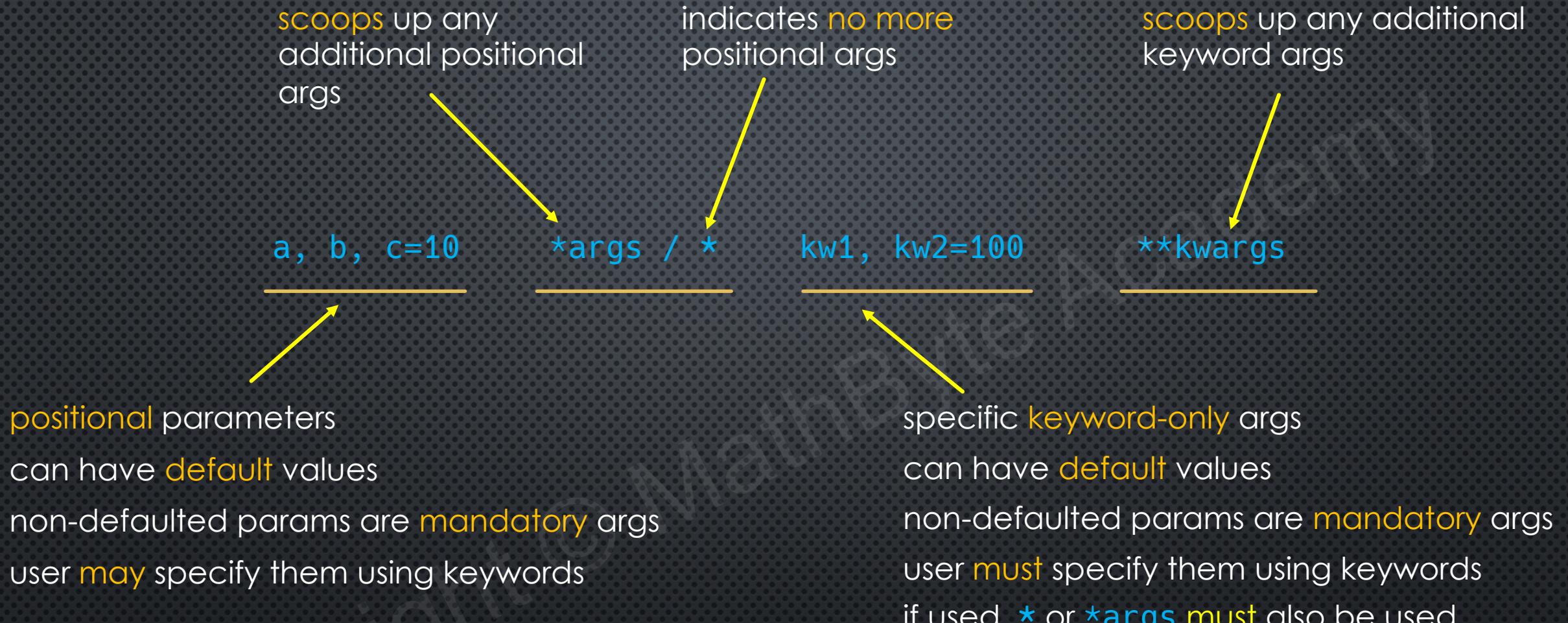
```
func()                  → args = ()  
                           kwargs = {}
```

PUTTING IT ALL TOGETHER

Copyright © 2014

Recap

positional arguments		keyword-only arguments	
specific	may have default values	after positional arguments have been exhausted	
*args	collects, and exhausts remaining positional arguments	specific	may have default values
*	indicates the end of positional arguments (effectively exhausts)	**kwargs	collects any remaining keyword arguments



Typical Use Case: Python's `print()` function

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argument is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument.

`*objects` arbitrary number of positional arguments

after that are keyword-only arguments

they all have default values, so they are all optional

Typical Use Cases

Often, keyword-only arguments are used to modify the default behavior of a function such as the `print()` function we just saw

```
def calc_hi_lo_avg(*args, log_to_console=False):
    hi = int(bool(args)) and max(args)
    lo = int(bool(args)) and min(args)
    avg = (hi + lo)/2
    if log_to_console:
        print("high={0}, low={1}, avg={2}".format(hi, lo, avg))
    return avg
```

Other times, keyword-only arguments might be used to make things clearer.

Having many positional parameters can become confusing, and extra care has to be taken to ensure the correct parameters are passed in the correct sequence.

DEFAULT VALUES



What happens at run-time...

When a module is loaded: all code is **executed** immediately

Module Code

a = 10

the integer object **10** is created and **a** references it

```
def func(a):  
    print(a)
```

the function object is created, and **func** references it

func(a)

the function is **executed**

What about default values?

Module Code

```
def func(a=10):  
    print(a)
```

```
func()
```

the **function** object is created, and **func** references it
the **integer** object **10** is evaluated/created
and is assigned as the default for **a**

the function is **executed**

by the time this happens, the default value for **a** has **already** been
evaluated and assigned – it is **not re-evaluated** when the function is
called



So what?

Consider this:

We want to create a function that will write a log entry to the console with a user-specified event date/time. If the user does not supply a date/time, we want to set it to the current date/time.

```
from datetime import datetime
```

```
def log(msg, *, dt=datetime.utcnow()):  
    print('{0}: {1}'.format(dt, msg))
```

```
log('message 1') → 2017-08-21 20:54:37.706994 : message 1
```

a few minutes later:

```
log('message 2') → 2017-08-21 20:54:37.706994 : message 2
```

Solution Pattern

We set a default for `dt` to `None`

Inside the function, we `test` to see if `dt` is still `None`

if `dt` is `None`, set it to the current date/time

otherwise, use what the caller specified for `dt`

recall that this is equivalent to:
`if not dt:`
 `dt = datetime.utcnow()`

```
from datetime import datetime  
  
def log(msg, *, dt=None):  
    dt = dt or datetime.utcnow()  
    print('{0}: {1}'.format(dt, msg))
```



In general, always beware of using a mutable object (or a callable) for an argument default

FIRST-CLASS FUNCTIONS

Copyright © 2018

First-Class Objects

can be passed to a function as an argument

can be returned from a function

can be assigned to a variable

can be stored in a data structure (such as list, tuple, dictionary, etc.)

Types such as `int`, `float`, `string`, `tuple`, `list` and many more are first-class objects.

Functions (`function`) are also first-class objects

Higher-Order Functions

Higher-order functions are functions that:

take a function as an argument (e.g. the simple timer we wrote in the last section)

and/or

return a function (plenty of that when we cover decorators in the next section)

Topics in this section

function annotations and documentation

lambda expressions and anonymous functions

callables

function introspection

built-in higher order functions (such as `sorted`, `map`, `filter`)

some functions in the `functools` module (such as `reduce`, `all`, `any`)

partials

DOCSTRINGS AND ANNOTATIONS

Copyright © 2018, DataCamp

Docstrings

We have seen the `help(x)` function before

→ returns some documentation (if available) for `x`

We can document our functions (and modules, classes, etc) to achieve the same result using docstrings
→ PEP 257

If the first line in the function body is a string (not an assignment, not a comment, just a string by itself), it will be interpreted as a docstring

```
def my_func(a):  
    "documentation for my_func"  
    return a
```

```
help(my_func)  
→ my_func(a)  
    documentation for my_func
```

Multi-line docstrings are achieved using...

multi-line strings!

Where are docstrings stored?

In the function's `__doc__` property

```
def fact(n):
    """Calculates n! (factorial function)
```

Inputs:

n: non-negative integer

Returns:

the factorial of n

"""

...

```
fact.__doc__ → 'Calculates n! (factorial function)\n\nInputs:\n\nn: non-negative integer\nReturns:\nthe\nfactorial of n\n'
```

```
help(fact) → fact(n)
Calculates n! (factorial function)
```

Inputs:

n: non-negative integer

Returns:

the factorial of n

Function Annotations

Function annotations give us an additional way to document our functions:

→ PEP 3107

```
def my_func(a: <expression>, b: <expression>) -> <expression>:  
    pass
```

```
def my_func(a: 'a string', b: 'a positive integer') -> 'a string':  
    return a * b
```

```
help(my_func) → my_func(a:'a string', b:'a positive integer') -> 'a string'
```

```
my_func.__doc__ → empty string
```


Default values, *args, **kwargs

can still be used as before

```
def my_func(a: str = 'xyz', b: int = 1) -> str:  
    pass
```

```
def my_func(a: str = 'xyz',  
           *args: 'additional parameters',  
           b: int = 1,  
           **kwargs: 'additional keyword only params') -> str:  
    pass
```

Where are annotations stored?

In the `__annotations__` property of the function

→ dictionary keys are the parameter names
 for a return annotation, the key is `return`
 values are the annotations

```
def my_func(a: 'info on a', b: int) -> float:  
    pass
```

```
my_func.__annotations__
```

→ `{'a': 'info on a', 'b': int, 'return': float}`

Where does Python use docstrings and annotations?

It doesn't really!

Mainly used by external tools and modules

Example: apps that generate documentation from your code (Sphinx)

Docstrings and annotations are entirely **optional**, and do not "force" anything in our Python code

We'll look at an enhanced version of annotations in an upcoming section on **type hints**

LAMBDA EXPRESSIONS

Copyright © 2014
Microsoft Corporation

What are Lambda Expressions?

We already know how to create functions using the `def` statement

Lambda expressions are simply another way to create functions

anonymous functions



the expression **returns** a function object
that evaluates and returns the **expression** when it is **called**

it can be assigned to a variable

passed as an argument to another function

it is a **function**, just like one created with `def`

Examples

```
lambda x: x**2
```

```
lambda x, y: x + y
```

```
lambda : 'hello'
```

```
lambda s: s[::-1].upper()
```

```
type(lambda x: x**2) → function
```

Note that these expressions are **function objects**, but are not "named"

→ **anonymous functions**

Lambdas, or anonymous functions, are NOT equivalent to closures

Assigning a Lambda to a Variable Name

```
my_func = lambda x: x**2
```

```
type(my_func) → function
```

```
my_func(3) → 9
```

```
my_func(4) → 16
```

identical to:

```
def my_func(x):  
    return x**2
```

```
type(my_func) → function
```

```
my_func(3) → 9
```

```
my_func(4) → 16
```

Passing as an Argument to another Function

```
def apply_func(x, fn):  
    return fn(x)
```

```
apply_func(3, lambda x: x**2)      → 9
```

```
apply_func(2, lambda x: x + 5)      → 7
```

```
apply_func('abc', lambda x: x[1:] * 3) → bcbcbc
```

equivalently:

```
def fn_1(x):  
    return x[1:] * 3
```

```
apply_func('abc', fn_1) → bcbcbc
```

Limitations

The "body" of a `lambda` is limited to a single expression

no assignments

```
lambda x: x = 5 
```

```
lambda x: x = x + 5 
```

no annotations

```
def func(x: int):  
    return x**2 
```

```
lambda x:int : x*2 
```

single logical line of code

→ line-continuation is OK, but still just one expression

```
lambda x: x * \  
math.sin(x) 
```

FUNCTION INTROSPECTION

Copyright © 2014

Functions are first-class objects

They have attributes

`__doc__`

`__annotations__`

We can attach our own attributes

```
def my_func(a, b):  
    return a + b
```

```
my_func.category = 'math'  
my_func.sub_category = 'arithmetic'
```

```
print(my_func.category)      → math
```

```
print(my_func.sub_category)  → arithmetic
```

The `dir()` function

`dir()` is a built-in function that, given an object as an argument, will return a list of valid attributes for that object

```
dir(my_func)
```

```
[('__annotations__', '__call__', '__class__', '__closure__',
 '__code__', '__defaults__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__get__', '__getattribute__', '__globals__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__kwdefaults__',
 '__le__', '__lt__', '__module__', '__name__',
 '__ne__', '__new__', '__qualname__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', 'category', 'sub_category']
```

Function Attributes: `__name__`, `__defaults__`, `__kwdefaults__`

`__name__` → name of function

`__defaults__` → tuple containing positional parameter defaults

`__kwdefaults__` → dictionary containing keyword-only parameter defaults

```
def my_func(a, b=2, c=3, *, kw1, kw2=2):  
    pass
```

`my_func.__name__` → `my_func`

`my_func.__defaults__` → `(2, 3)`

`my_func.__kwdefaults__` → `{'kw2': 2}`

Function Attribute: `__code__`

```
def my_func(a, b=1, *args, **kwargs):  
    i = 10  
    b = min(i, b)  
    return a * b
```

`my_func.__code__`
→ <code object my_func at 0x00020EEF ... >

This `__code__` object itself has various properties, which include:

`co_varnames` parameter and local variables

```
my_func.__code__.co_varnames → ('a', 'b', 'args', 'kwargs', 'i')
```

parameter names **first**, followed by local variable names

`co_argcount` number of parameters

```
my_func.__code__.co_argcount → 2
```

does not count `*args` and `**kwargs`!

The `inspect` Module

`import inspect`

`ismethod(obj)` `isfunction(obj)` `isroutine(obj)`

and many others...

What's the difference between a `function` and a `method`?

Classes and objects have `attributes` – an object that is bound (to the class or the object)

An attribute that is `callable`, is called a `method`

```
def my_func():  
    pass
```

`func` is bound to `my_obj`, an instance of `MyClass`

```
def MyClass:  
    def func(self):  
        pass
```

`isfunction(my_func) → True`

```
my_obj = MyClass()
```

`ismethod(my_func) → False`

`isfunction(my_obj.func) → False`

`ismethod(my_obj.func) → True`

`isroutine(my_func) → True`

`isroutine(my_obj.func) → True`

Code Introspection

We can recover the source code of our functions/methods

`inspect.getsource(my_func)` → a string containing our entire def statement, including annotations, docstrings, etc

We can find out in which module our function was created

`inspect.getmodule(my_func)` → <module '__main__'>

`inspect.getmodule(print)` → <module 'builtins' (built-in)>

`inspect.getmodule(math.sin)` → <module 'math' (built-in)>

Function Comments

```
# setting up variable  
i = 10  
  
# TODO: Implement function  
# some additional notes  
def my_func(a, b=1):  
    # comment inside my_func  
    pass
```

```
inspect.getcomments(my_func)
```

```
→ '# TODO: Implement function\n# some additional notes'
```

Many IDE's support the TODO comment to flag functions and other callables

Note that this is not the same as docstrings

Callable Signatures

`inspect.signature(my_func)` → `Signature` instance

Contains an attribute called `parameters`

Essentially a dictionary of parameter names (keys), and metadata about the parameters (values)

keys → parameter name

values → object with attributes such as `name`, `default`, `annotation`, `kind`

`kind` `POSITIONAL_OR_KEYWORD`

`VAR_POSITIONAL`

`KEYWORD_ONLY`

`VAR_KEYWORD`

`POSITIONAL_ONLY`

Callable Signatures

```
def my_func(a: 'a string',
            b: int = 1,
            *args: 'additional positional args',
            kw1: 'first keyword-only arg',
            kw2: 'second keyword-only arg' = 10,
            **kwargs: 'additional keyword-only args') -> str:
    """does something
       or other"""
    pass
```

```
for param in inspect.signature(my_func).parameters.values():
    print('Name:', param.name)
    print('Default:', param.default)
    print('Annotation:', param.annotation)
    print('Kind:', param.kind)
```

CALLABLES

Copyright © 2014

What are callables?

any object that can be called using the () operator

callables **always** return a value

like **functions** and **methods**

but it goes beyond just those two...

many other objects in Python are also callable

To see if an object is callable, we can use the built-in function: **callable**

```
callable(print) → True
```

```
callable('abc'.upper) → True
```

```
callable(str.upper) → True
```

```
callable(callable) → True
```

```
callable(10) → False
```

Different Types of Callables

built-in functions

`print` `len` `callable`

built-in methods

`a_str.upper` `a_list.append`

user-defined functions

created using `def` or `lambda` expressions

methods

functions bound to an object

classes

`MyClass(x, y, z)`

→ `__new__(x, y, z)` → creates the new object

→ `__init__(self, x, y, z)`

→ returns the object (reference)

class instances

if the class implements `__call__` method

generators, coroutines, asynchronous generators

MAP, FILTER, ZIP

Higher order functions

A function that takes a function as a parameter and/or returns a function as its return value

Example: `sorted`

`map`
`filter`

}

modern alternative → list comprehensions and generator expressions

The `map` function

`map(func, *iterables)`

`*iterables` → a variable number of iterable objects

`func` → some function that takes as many arguments as there are `iterable` objects passed to `iterables`

`map(func, *iterables)` will then return an `iterator` that calculates the function applied to each element of the iterables

The iterator stops as soon as one of the iterables has been exhausted so, unequal length iterables can be used

Examples

```
l = [2, 3, 4]
```

```
def sq(x):  
    return x**2
```

```
list(map(sq, l))      → [4, 9, 16]
```

```
l1 = [1, 2, 3]
```

```
l2 = [10, 20, 30]
```

```
def add(x, y):  
    return x + y
```

```
list(map(add, l1, l2))  → [11, 22, 33]
```

```
list(map(lambda x, y: x + y, l1, l2))  → [11, 22, 33]
```

The `filter` function

`filter(func, iterable)`

`iterable` → a single iterable

`func` → some function that takes a single argument

`filter(func, iterable)` will then return an iterator that contains all the elements of the iterable for which the function called on it is Truthy

If the function is `None`, it simply returns the elements of `iterable` that are Truthy

Examples

```
l = [0, 1, 2, 3, 4]
```

```
list(filter(None, l)) → [1, 2, 3, 4]
```

```
def is_even(n):  
    return n % 2 == 0
```

```
list(filter(is_even, l)) → [0, 2, 4]
```

```
list(filter(lambda n: n % 2 == 0, l)) → [0, 2, 4]
```

The `zip` function

`zip(*iterables)`

[1, 2, 3, 4] $\xrightarrow{\text{zip}}$ (1, 10), (2, 20), (3, 30), (4, 40)
[10, 20, 30, 40]

[1, 2, 3]
[10, 20, 30] $\xrightarrow{\text{zip}}$ (1, 10, 'a'), (2, 20, 'b'), (3, 30, 'c')
['a', 'b', 'c']

[1, 2, 3, 4, 5] $\xrightarrow{\text{zip}}$ (1, 10), (2, 20), (3, 30)
[10, 20, 30]

Examples

```
l1 = [1, 2, 3]  
l2 = [10, 20, 30, 40]  
l3 = 'python'
```

```
list(zip(l1, l2, l3)) → [(1, 10, 'p'), (2, 20, 'y'), (3, 30, 't')]
```

```
l1 = range(100)  
l2 = 'abcd'
```

```
list(zip(l1, l2)) → [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

List Comprehension Alternative to map

```
l = [2, 3, 4]
```

```
def sq(x):  
    return x**2  
list(map(sq, l)) ] list(map(lambda x: x**2, l)) → [4, 9, 16]
```

```
result = []  
for x in l:  
    result.append(x**2)    result → [4, 9, 16]
```

```
[x**2 for x in l] → [4, 9, 16]
```

```
[<expression> for <varname> in <iterable>]
```

List Comprehension Alternative to `map`

```
l1 = [1, 2, 3]
l2 = [10, 20, 30]
```

```
list(map(lambda x, y: x + y, l1, l2)) → [11, 22, 33]
```

Remember: `zip(l1, l2)` → `[(1, 10), (2, 20), (3, 30)]`

```
[x + y for x, y in zip(l1, l2)] → [11, 22, 33]
```


Combining `map` and `filter`

```
l = range(10)  
list(filter(lambda y: y < 25, map(lambda x: x**2, l))) → [0, 1, 4, 9, 16]
```

Using a list comprehension is much clearer:

```
[x**2 for x in range(10) if x**2 < 25] → [0, 1, 4, 9, 16]
```

REDUCING FUNCTIONS

Copyright © 2014

Reducing Functions in Python

These are functions that recombine an iterable recursively, ending up with a single return value

Also called **accumulators**, **aggregators**, or **folding functions**.

Example: Finding the maximum value in an iterable

$a_0, a_1, a_2, \dots, a_{n-1}$

`max(a, b)` → maximum of **a** and **b**

`result = a0`

`result = max(result, a1)`

`result = max(result, a2)`

...

`result = max(result, an-1)`

→ max value in $a_0, a_1, a_2, \dots, a_{n-1}$

Because we have not studied iterables in general, we will stay with the special case of sequences.
(i.e. we can use indexes to access elements in the sequence)

Using a loop

```
l = [5, 8, 6, 10, 9]
```

```
max_value = lambda a, b: a if a > b else b
```

```
def max_sequence(sequence):
    result = sequence[0]
    for e in sequence[1:]:
        result = max_value(result, e)
    return result
```

```
result = 5
```

```
result = max(5, 8) = 8
```

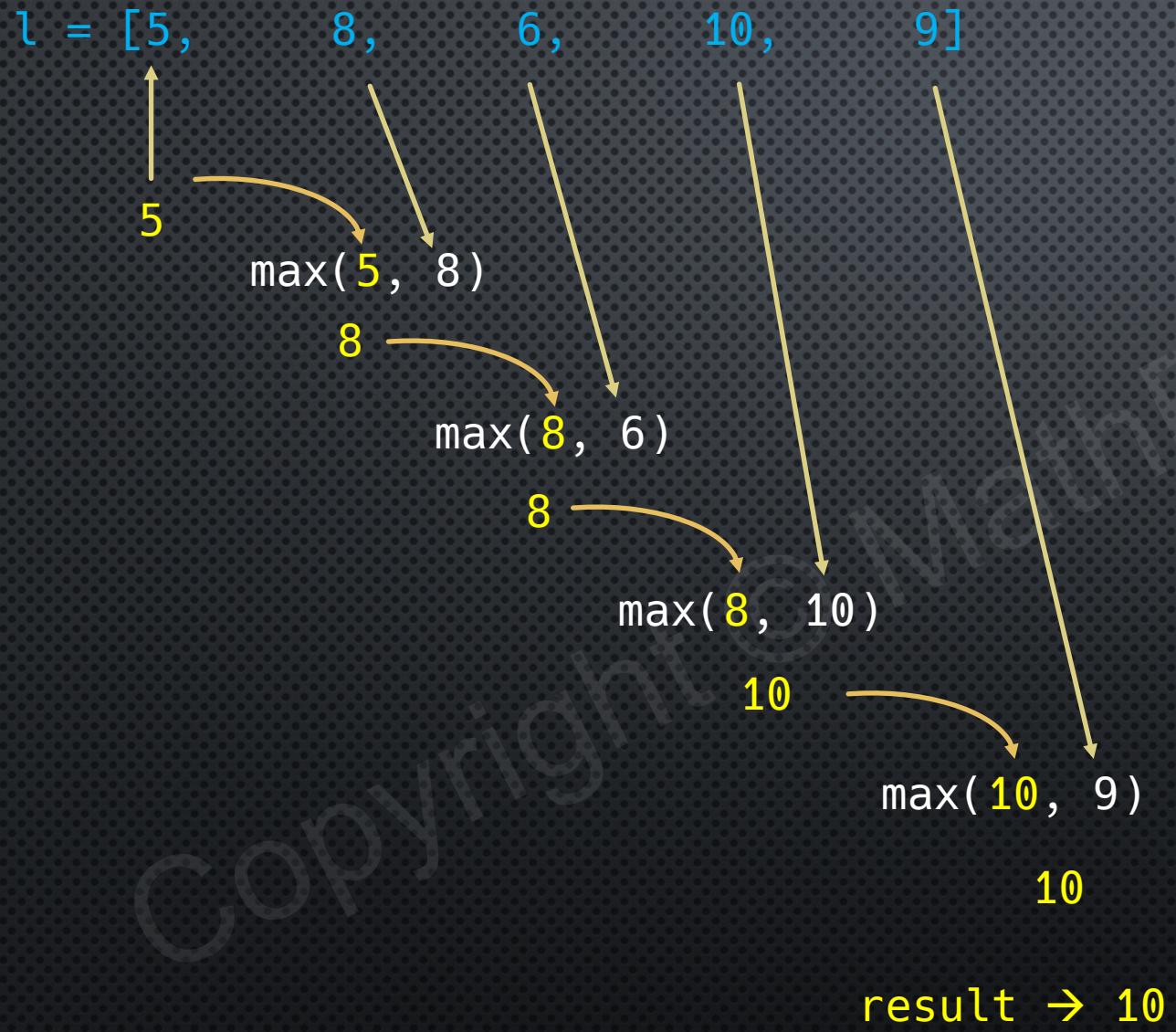
```
result = max(8, 6) = 8
```

```
result = max(8, 10) = 10
```

```
result = max(10, 9) = 10
```

result → 10

Notice the sequence of steps:



To calculate the min:

```
l = [5, 8, 6, 10, 9]
```

```
min_value = lambda a, b: a if a < b else b
```

```
def min_sequence(sequence):
    result = sequence[0]
    for e in sequence[1:]:
        result = min_value(result, e)
    return result
```

In fact we could write:

```
def _reduce(fn, sequence):
    result = sequence[0]
    for x in sequence[1:]:
        result = fn(result, x)
    return result
```

```
_reduce(lambda a, b: a if a > b else b, l) → maximum
```

```
_reduce(lambda a, b: a if a < b else b, l) → minimum
```

All we really needed to do was to change the function that is repeatedly applied.

Adding all the elements in a list

```
add = lambda a, b: a+b
```

```
l = [5, 8, 6, 10, 9]
```

```
def _reduce(fn, sequence):
    result = sequence[0]
    for x in sequence[1:]:
        result = fn(result, x)
    return result
```

```
_reduce(add, l)
```

result = 5

result = add(5, 8) = 13

result = add(13, 6) = 19

result = add(19, 10) = 29

result = add(29, 9) = 38

result → 38

The `functools` module

Python implements a `reduce` function that will handle any iterable, but works similarly to what we just saw

```
from functools import reduce
```

```
l = [5, 8, 6, 10, 9]
```

```
reduce(lambda a, b: a if a > b else b, l) → max → 10
```

```
reduce(lambda a, b: a if a < b else b, l) → min → 5
```

```
reduce(lambda a, b: a + b, l) → sum → 38
```

reduce works on any iterable

```
reduce(lambda a, b: a if a < b else b, {10, 5, 2, 4}) → 2
```

```
reduce(lambda a, b: a if a < b else b, 'python') → h
```

```
reduce(lambda a, b: a + ' ' + b, ('python', 'is', 'awesome!'))  
→ 'python is awesome'
```

Built-in Reducing Functions

Python provides several common reducing functions:

min `min([5, 8, 6, 10, 9])` → 5

max `max([5, 8, 6, 10, 9])` → 10

sum `sum([5, 8, 6, 10, 9])` → 38

any `any(l)` → **True** if any element in `l` is truthy
 False otherwise

all `all(l)` → **True** if every element in `l` is truthy
 False otherwise

Using `reduce` to reproduce `any`

```
l = [0, '', None, 100]
```

```
result = bool(0) or bool('') or bool(None) or bool(100)
```

Note: `0` or `''` or `None` or `100` → `100` but we want our result to be True/False so we use `bool()`

Here we just need to repeatedly apply the `or` operator to the truth values of each element

```
result = bool(0) → False
```

```
result = result or bool('') → False
```

```
result = result or bool(None) → False
```

```
result = result or bool(100) → True
```

```
reduce(lambda a, b: bool(a) or bool(b), l) → True
```

Calculating the product of all elements in an iterable

No built-in method to do this

But very similar to how we added all the elements in an iterable or sequence:

[1, 3, 5, 6] → 1 * 3 * 5 * 6

`reduce(lambda a, b: a * b, l)`

res = 1

res = res * 3 = 3

res = res * 5 = 3 * 5 = 15

res = res * 6 = 15 * 6 = 90 = 1 * 3 * 5 * 6

Special case: Calculating $n!$

$$n! = 1 * 2 * 3 * \dots * n$$

$$5! = 1 * 2 * 3 * 4 * 5$$

`range(1, 6)` $\rightarrow 1, 2, 3, 4, 5$

`range(1, n+1)` $\rightarrow 1, 2, 3, \dots, n$

To calculate `n!` we need to find the product of all the elements in `range(1, n+1)`

`reduce(lambda a, b: a * b, range(1, 5+1))` $\rightarrow 5!$

The `reduce` initializer

The `reduce` function has a third (optional) parameter: `initializer` (defaults to `None`)

If it is specified, it is essentially like adding it to the front of the iterable.

It is often used to provide some kind of default in case the iterable is empty.

```
l = []
reduce(lambda x, y: x+y, l)      → exception
```

```
l = []
reduce(lambda x, y: x+y, l, 1)    → 1
```

```
l = [1, 2, 3]
reduce(lambda x, y: x+y, l, 1)    → 7
```

```
l = [1, 2, 3]
reduce(lambda x, y: x+y, l, 100)  → 106
```

PARTIAL FUNCTIONS

Reducing Function Arguments

```
def my_func(a, b, c):  
    print(a, b, c)
```

```
def fn(b, c):  
    return my_func(10, b, c)
```

fn(20, 30) → 10, 20, 30

```
f = lambda b, c: my_func(10, b, c)
```

f(20, 30) → 10, 20, 30

```
from functools import partial
```

```
f = partial(my_func, 10)
```

f(20, 30) → 10, 20, 30

Handling more complex arguments

```
def pow(base, exponent):  
    return base ** exponent
```

```
square = partial(pow, exponent=2)  
cube = partial(pow, exponent=3)
```

```
square(5)      → 25
```

```
cube(5)       → 75
```

```
cube(base=5)  → 75
```

!! square(5, exponent=3) → 75

Beware!!

You can use variables when creating partials

but there arises a similar issue to argument default values

```
def my_func(a, b, c):  
    print(a, b, c)
```

```
a = 10  
f = partial(my_func, a)
```

f(20, 30) → 10, 20, 30

a = 100

f(20, 30) → 10, 20, 30

the memory address of 10 is baked in to the partial

a now points to a different memory address

but the partial still points to the original object (10)

If a is mutable (e.g. a list), then its contents can be changed

THE operator MODULE

Copyright © 2014
McGraw-Hill Education

Functional Equivalents to Operators

In the last lecture we wrote code such as:

```
l = [2, 3, 4]
```

```
reduce(lambda a, b: a * b, l)
```

We used a lambda expression to create a functional version of the `*` operator

This is something that happens quite often, so the `operator` module was created

This module is a convenience module.

You can always use your own functions and lambda expressions instead.

The operator module

Arithmetic Functions

`add(a, b)`

`mul(a, b)`

`pow(a, b)`

`mod(a, b)`

`floordiv(a, b)`

`neg(a)`

and many more...

Sequence/Mapping Operators

`concat(s1, s2)`

`contains(s, val)`

`countof(s, val)`

`getitem(s, i)`

`setitem(s, i, val)`

`delitem(s, i)`

mutable objects

variants that use `slices`

Item Getters

The `itemgetter` function returns a **callable**

`getitem(s, i)` takes two parameters, and returns a value: `s[i]`

```
s = [1, 2, 3]
```

```
getitem(s, 1) → 2
```

`itemgetter(i)` returns a **callable** which takes one parameter: a sequence object

```
f = itemgetter(1)
```

```
s = [1, 2, 3]
```

```
f(s) → 2
```

```
s = 'python'
```

```
f(s) → 'y'
```

Item Getters

We can pass more than one index to `itemgetter`:

```
l = [1, 2, 3, 4, 5, 6]
s = 'python'
```

```
f = itemgetter(1, 3, 4)
```

```
f(l) → (2, 4, 5)
```

```
f(s) → ('y', 'h', 'o')
```

Attribute Getters

The `attrgetter` function is similar to `itemgetter`, but is used to retrieve `object` attributes

It also returns a `callable`, that takes the object as an argument

Suppose `my_obj` is an object with three properties:

```
my_obj.a → 10  
my_obj.b → 20  
my_obj.c → 30
```

```
f = attrgetter('a')      f(my_obj) → 10
```

```
f = attrgetter('a', 'c')    f(my_obj) → (10, 30)
```

Can also call directly:

```
attrgetter('a', 'b', 'c')(my_obj) → (10, 20, 30)
```

Calling another Callable

Consider the `str` class that provides the `upper()` method:

`s = 'python'` `s.upper() → PYTHON`

`f = attrgetter('upper')`

`f(s)` → returns the `upper` method of `s`
it is a **callable**, and can be called using `()`

`f(s)()` → PYTHON

`attrgetter('upper')(s)()` → PYTHON

Or, we can use the slightly simpler `methodcaller` function

`methodcaller('upper')('python')` → PYTHON

Basically, `methodcaller` retrieves the named attribute **and** calls it as well

It can also handle more arguments, as we'll see in the code

SCOPES, CLOSURES AND DECORATORS

Copyright © 2018, Manning Publications Company

Variable Scopes

local scope
global scope
nonlocal scope
nested scopes

Closures

what they are
closure scopes
(they are not equivalent to lambdas!)

Decorators

what they are
how they are related to closures
convenience of using @

Applications

GLOBAL AND LOCAL SCOPES

Copyright ©

Scopes and Namespaces

When an object is assigned to a variable `a = 10`

that variable points to some object

and we say that the variable (name) is **bound** to that object

That object can be accessed using that name in various parts of our code

But not just anywhere!

That variable name and it's binding (name and object) only "exist" in specific parts of our code

the portion of code where that name/binding is defined, is called the **lexical scope** of the variable

these bindings are stored in **namespaces**

(each scope has its own namespace)

The Global Scope

The **global** scope is essentially the **module** scope.

It spans a **single** file only.

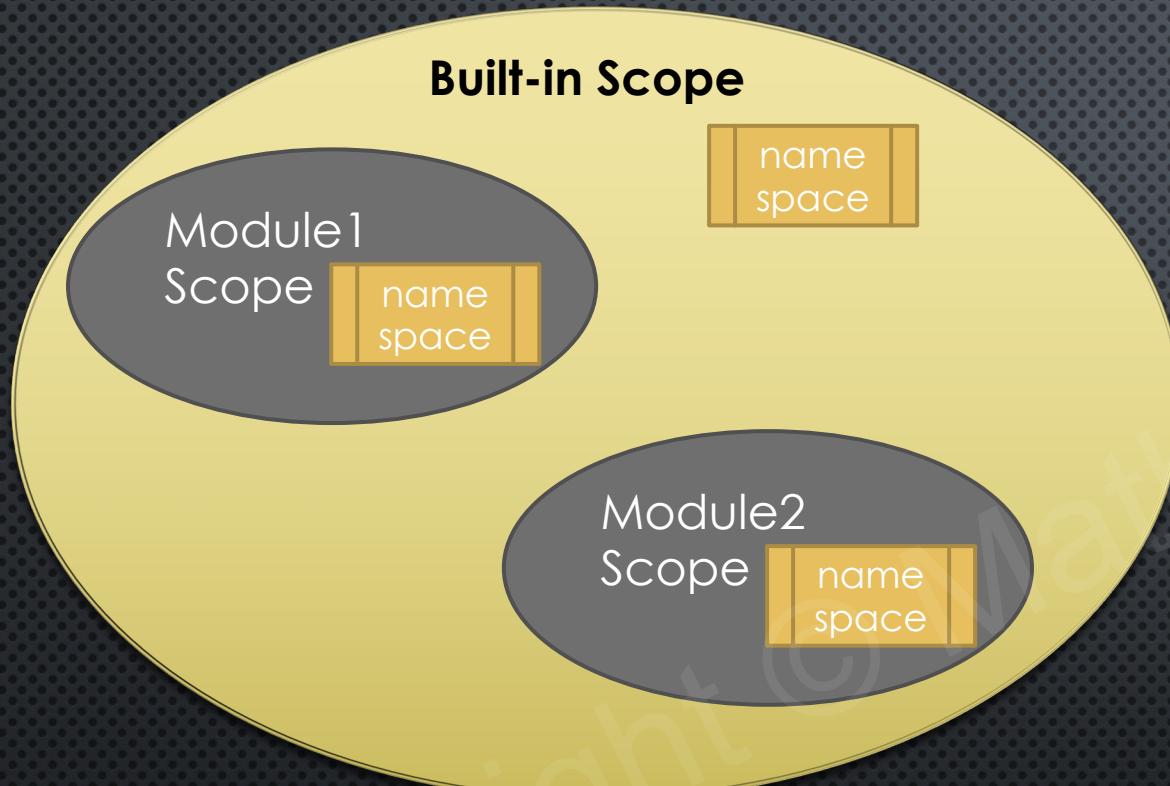
There is no concept of a truly global (across all the modules in our entire app) scope in Python.

The only exception to this are some of the **built-in** globally available objects, such as:

`True` `False` `None` `dict` `print`

The built-in and global variables can be used **anywhere** inside our module
including inside any **function**

Global scopes are nested inside the built-in scope



name space	
var1	0xA345E
func1	0xFF34A

If you reference a variable name inside a scope and Python does not find it in that scope's namespace it will look for it in an enclosing scope's namespace

Examples

module1.py

```
print(True)
```

Python does not find `True` or `print` in the current (module/global) scope
So, it looks for them in the enclosing scope → built-in
Finds them there → `True`

module2.py

```
print(a)
```

Python does not find `a` or `print` in the current (module/global) scope
So, it looks for them in the enclosing scope → built-in
Find `print`, but not `a` → run-time Name Error

module3.py

```
print = lambda x: 'hello {}'.format(x)
```

```
s = print('world')
```

Python finds `print` in the module scope
So it uses it!
`s` → `hello world!`

The Local Scope

When we create functions, we can create variable names inside those functions (using assignments)

e.g. `a = 10`

Variables defined inside a function are not created until the function is **called**

Every time the function is called, a **new scope** is created

Variables defined inside the function are assigned to that scope

→ Function Local scope

→ Local scope

The actual object the variable references could be **different** each time the function is called

(this is why recursion works!)

Example

```
def my_func(a, b):  
    c = a * b  
    return c
```

```
my_func('z', 2)
```

```
my_func(10, 5)
```

my_func

a

b

c

these names will be considered local
to **my_func**

my_func

a → 'z'

b → 2

c → 'zz'

same names, different local scopes

my_func

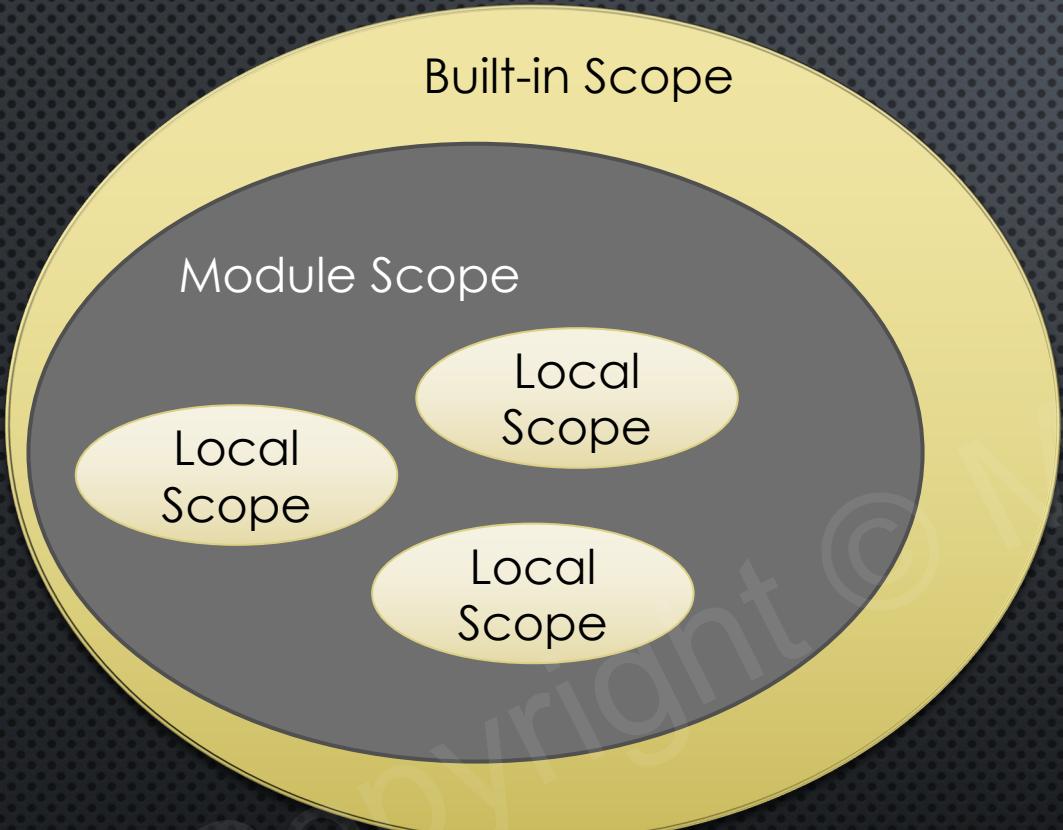
a → 10

b → 5

c → 50

Nested Scopes

Scopes are often nested



Namespace lookups

When requesting the object bound to a variable name:

e.g. `print(a)`

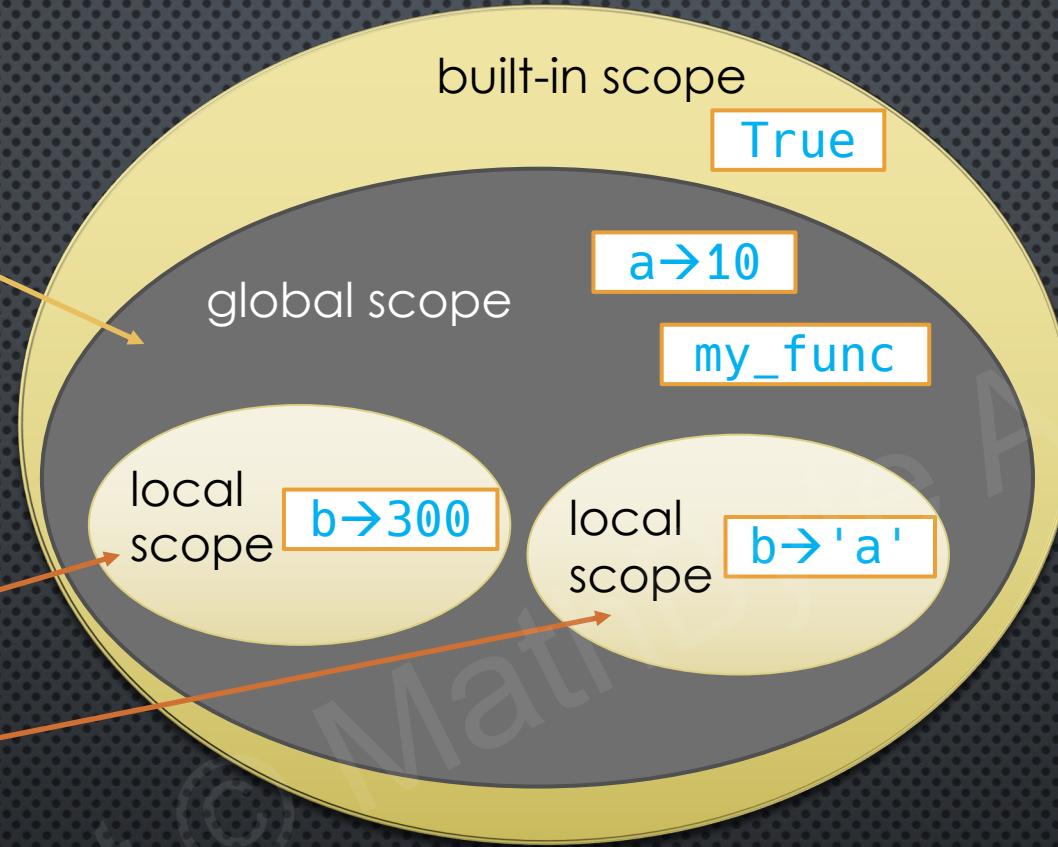
Python will try to find the object bound to the variable

- in current local scope first
- works up the chain of enclosing scopes

Example

module1.py

```
a = 10  
def my_func(b):  
    print(True)  
    print(a)  
    print(b)  
  
my_func(300)  
  
my_func('a')
```



Remember reference counting?

When `my_func(var)` finishes running, the scope is gone too!

and the reference count of the object `var` was bound to (referenced) is decremented

We also say that `var` goes out of scope

Accessing the global scope from a local scope

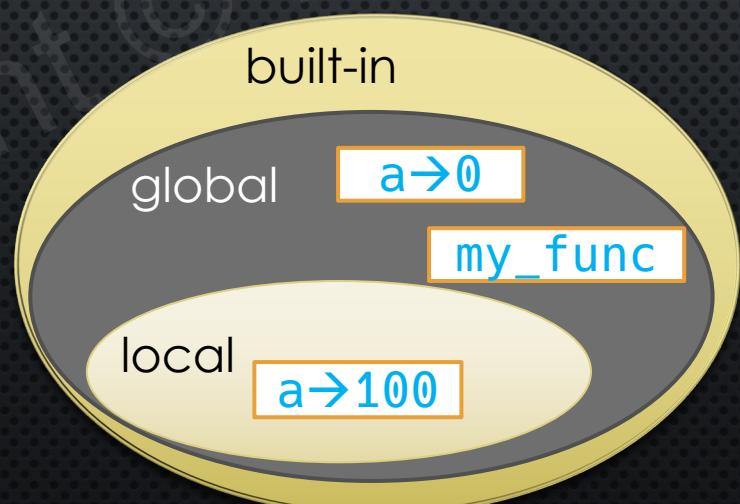
When **retrieving** the value of a global variable from inside a function, Python automatically searches the local scope's namespace, and up the chain of all enclosing scope namespaces

local → global → built-in

What about modifying a global variables value from inside the function?

```
a = 0  
def my_func():  
    a = 100  
    print(a)  
  
my_func() → 100  
  
print(a) → 0
```

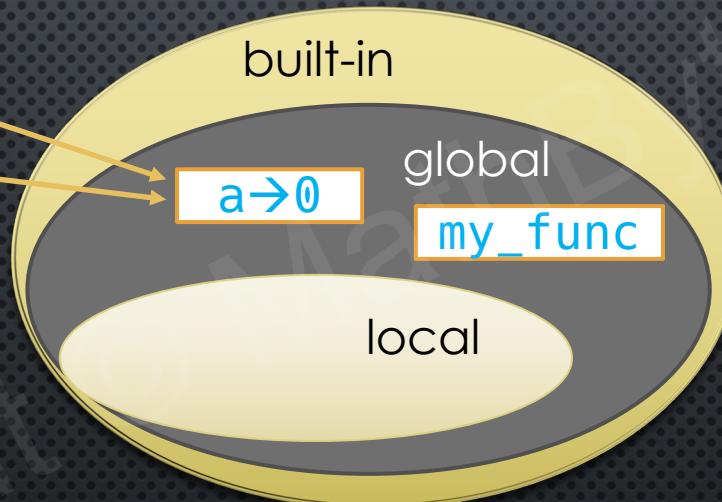
assignment → Python interprets this as a **local** variable (at compile-time)
→ the local variable **a** masks the global variable **a**



The `global` keyword

We can tell Python that a variable is meant to be scoped in the global scope by using the `global` keyword

```
a = 0  
def my_func():  
    global a  
    a = 100  
  
my_func()  
  
print(a) → 100
```



Example

```
counter = 0

def increment():
    global counter
    counter += 1

increment()
increment()
increment()

print(counter) → 3
```

Global and Local Scoping

When Python encounters a function definition at **compile-time**

it will scan for any labels (variables) that have values **assigned** to them (**anywhere** in the function)
if the label has not been specified as **global**, then it will be **local**

variables that are referenced but **not assigned** a value **anywhere** in the function will **not be local**,
and Python will, at **run-time**, look for them in enclosing scopes

```
a = 10
```

```
def func1():  
    print(a)
```

a is referenced only in entire function
at compile time → **a** non-local

```
def func2():  
    a = 100
```

assignment
at compile time → **a** local

```
def func3():  
    global a  
    a = 100
```

assignment
at compile time → **a** global
(because we told Python **a** was global)

```
def func4():  
    print(a)  
    a = 100
```

assignment
at compile time → **a** local

→ when we call **func4()**
print(a) results in a **run-time** error
because **a** is local, and we are
referencing it **before** we have
assigned a value to it!

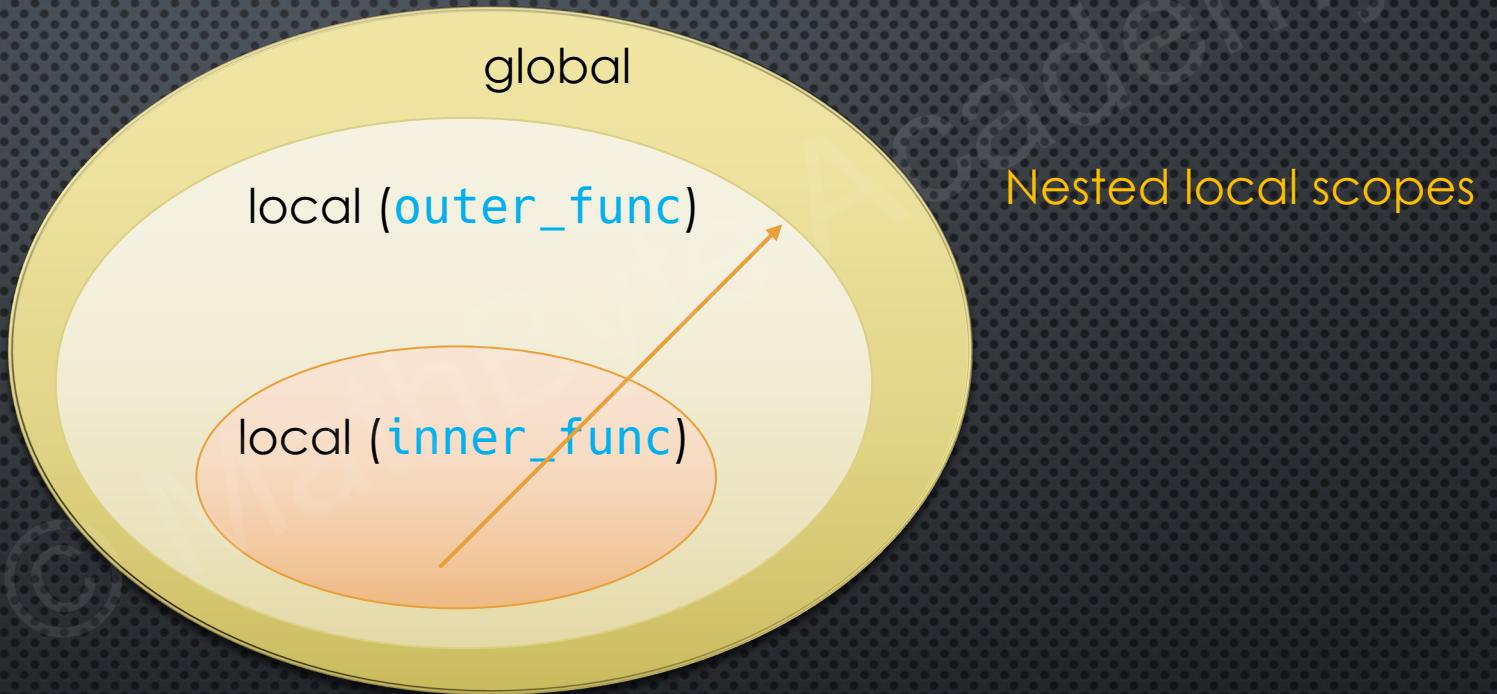
NONLOCAL SCOPES

Copyright © 2018, Dr. Christiano

Inner Functions

We can define functions from inside another function:

```
def outer_func():  
    # some code  
  
    def inner_func():  
        # some code  
  
        inner_func()  
  
outer_func()
```



Both functions have access to the global and built-in scopes as well as their respective local scopes

But the **inner** function also has access to its **enclosing** scope – the scope of the **outer** function

That scope is neither local (to **inner_func**) nor global – it is called a **nonlocal** scope

Referencing variables from the enclosing scope

Consider this example we have seen before:

```
module1.py  
a = 10  
  
def outer_func():  
    print(a)  
  
outer_func()
```

When we call `outer_func`, Python sees the reference to `a`.
Since `a` is not in the local scope, Python looks in the `enclosing` (global) scope

Referencing variables from the enclosing scope

Now consider this example:

module1.py

```
def outer_func():
    a = 10

    def inner_func():
        print(a)

    inner_func()

outer_func()
```

When we call `outer_func`, `inner_func` is created and called

When `inner_func` is called, Python does not find `a` in the local (`inner_func`) scope

So it looks for it in the `enclosing` scope, in this case the scope of `outer_func`

Referencing variables from the enclosing scope

```
module1.py
```

```
a = 10
```

```
def outer_func():
```

```
    def inner_func():
```

```
        print(a)
```

```
    inner_func()
```

```
outer_func()
```

When we call `outer_func`, `inner_func` is defined and called

When `inner_func` is called, Python does not find `a` in the local (`inner_func`) scope

So it looks for it in the `enclosing` scope, in this case the scope of `outer_func`

Since it does not find it there either, it looks in the `enclosing` (global) scope

Modifying global variables

We saw how to use the `global` keyword in order to modify a global variable within a nested scope

```
a = 10

def outer_func1():
    global a
    a = 1000

outer_func1()
print(a)      → 1000
```

We can of course do the same thing from within a nested function

```
def outer_func2():
    def inner_func():
        global a
        a = 'hello'
    inner_func()

outer_func2()
print(a)      → hello
```

Modifying nonlocal variables

Can we modify variables defined in the outer nonlocal scope?

```
def outer_func():
    x = 'hello'

def inner_func():
    x = 'python'

inner_func()
print(x)
outer_func() → hello
```

When `inner_func` is compiled, Python sees an assignment to `x`

So it determines that `x` is a local variable to `inner_func`

The variable `x` in `inner_func` masks the variable `x` in `outer_func`

Modifying nonlocal variables

Just as with global variables, we have to explicitly tell Python we are modifying a nonlocal variable

We can do that using the `nonlocal` keyword

```
def outer_func():
    x = 'hello'

    def inner_func():
        nonlocal x
        x = 'python'

    inner_func()

    print(x)

outer_func() → python
```

Nonlocal Variables

Whenever Python is told that a variable is **nonlocal**

it will look for it in the **enclosing local scopes** chain until it **first** encounters the specified variable name

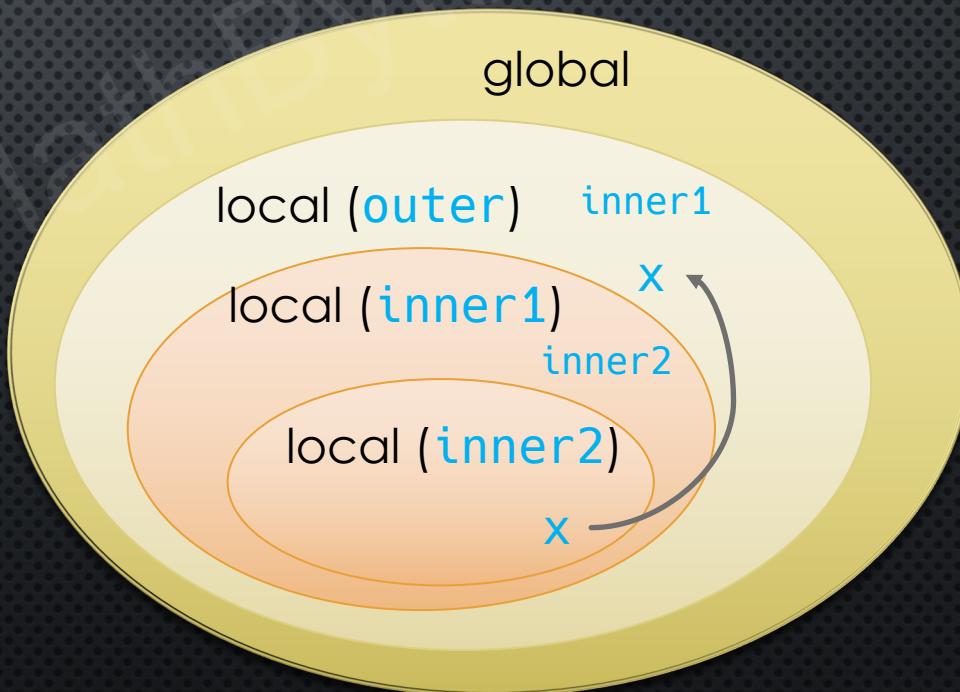
Beware: It will only look in local scopes, it will **not** look in the **global** scope

```
def outer():
    x = 'hello'

def inner1():
    def inner2():
        nonlocal x
        x = 'python'
    inner2()
inner1()
print(x)

outer()
```

→ python



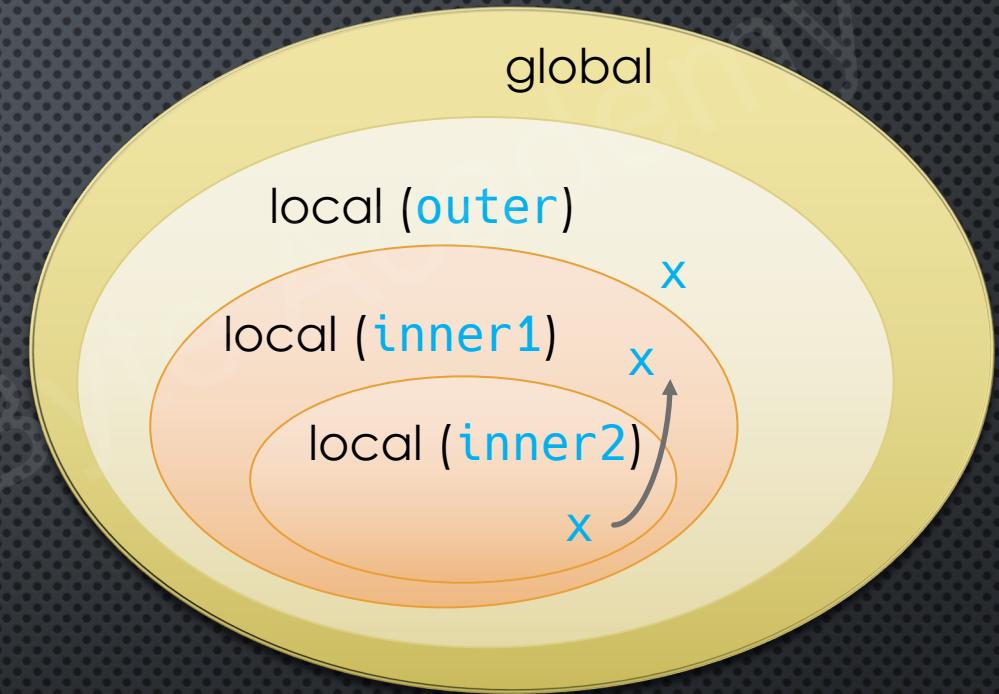
Nonlocal Variables

But consider this example:

```
def outer():
    x = 'hello'

def inner1():
    x = 'python'
    def inner2():
        nonlocal x
        x = 'monty'
    print('inner(before)', x)      → python
    inner2()
    print('inner(after)', x)      → monty

inner1()
print('outer', x)              → hello
outer()
```



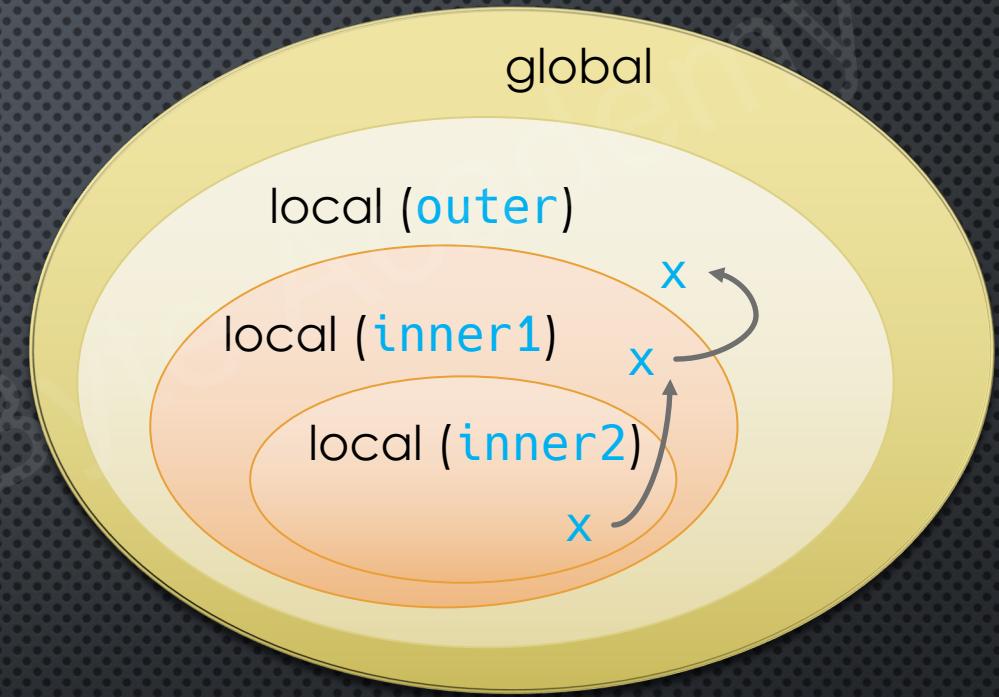
Nonlocal Variables

```
def outer():
    x = 'hello'

def inner1():
    nonlocal x
    x = 'python'
    def inner2():
        nonlocal x
        x = 'monty'
    print('inner(before)', x) → python
    inner2()
    print('inner(after)', x) → monty

inner1()
print('outer', x) → monty

outer()
```



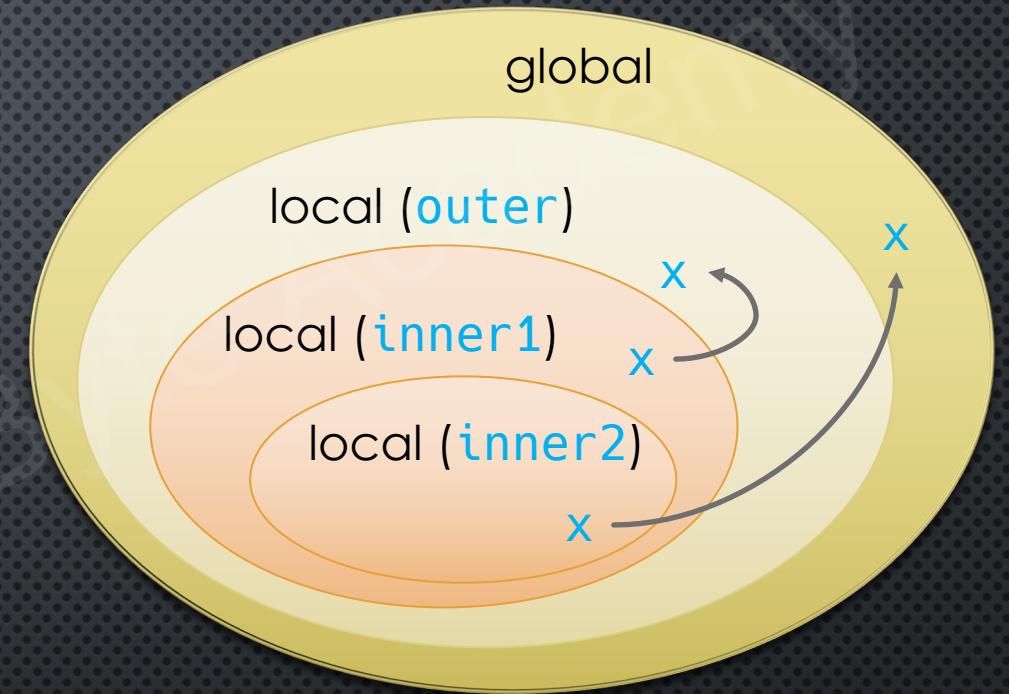
Nonlocal and Global Variables

```
x = 100
def outer():
    x = 'python'

    def inner1():
        nonlocal x
        x = 'monty'
        def inner2():
            global x
            x = 'hello'
            print('inner(before)', x) → monty
        inner2()
        print('inner(after)', x) → monty

    inner1()
    print('outer', x) → monty

outer()
print(x) → hello
```



CLOSURES

Copyright © 2014 by  NCS Pearson, Inc.

Free Variables and Closures

Remember: Functions defined inside another function can access the outer (nonlocal) variables

```
def outer():
    x = 'python'

    def inner():
        print("{0} rocks!".format(x))

    inner()

outer()
```

→ python rocks!

this **x** refers to the one in **outer**'s scope
this nonlocal variable **x** is called a **free** variable
when we consider **inner**, we really are looking at:

- the function **inner**
- the free variable **x** (with current value **python**)

This is called a **closure**

Returning the inner function

What happens if, instead of calling (running) `inner` from inside `outer`, we **return** it?

```
def outer():  
    x = 'python'  
  
    def inner():  
        print("{0} rocks!".format(x))  
  
    return inner
```

`x` is a free variable in `inner`
it is bound to the variable `x` in `outer`
this happens when `outer` runs
(i.e. when `inner` is created)
this is the **closure**

when we return `inner`, we are actually "returning" the **closure**

We can assign that return value to a variable name: `fn = outer()`

`fn()` → python rocks!

When we **called** `fn`

at that time Python determined the value of `x` in the extended scope

But notice that `outer` had finished running **before** we called `fn` – its scope was "gone"

Python Cells and Multi-Scoped Variables

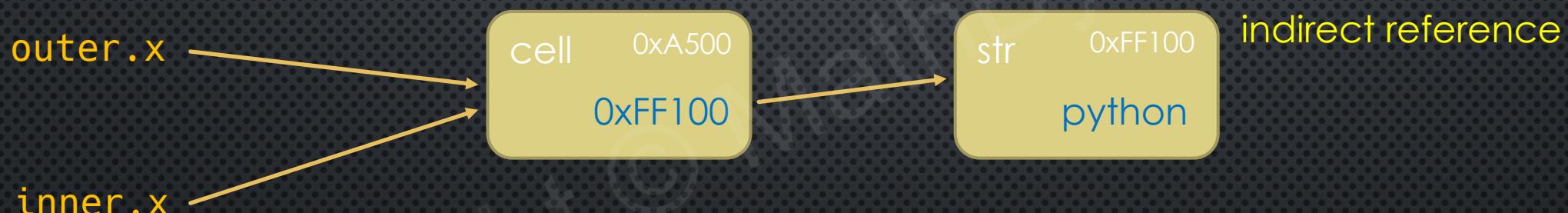
```
def outer():  
    x = 'python'  
    def inner():  
        print(x)  
    return inner
```

Here the value of `x` is shared between two scopes:

- `outer`
- `closure`

The label `x` is in two different scopes but always reference the same "value"

Python does this by creating a `cell` as an intermediary object



In effect, both variables `x` (in `outer` and `inner`), point to the same cell

When requesting the value of the variable, Python will "double-hop" to get to the final value

Closures

You can think of the closure as a function plus an extended scope that contains the free variables

The free variable's value is the object the cell points to – so that could change over time!

Every time the function in the closure is called and the free variable is referenced:

Python looks up the cell object, and then whatever the cell is pointing to

```
def outer():
    a = 100
    x = 'python'
    def inner():
        a = 10 # local variable
        print("{0} rocks!".format(x))
    return inner
fn = outer()
```

fn → inner + extended scope

closure

cell 0xA500
0xFF100

str 0xFF100
python

indirect reference



Introspection

```
def outer():
    a = 100
    x = 'python'
    def inner():
        a = 10 # local variable
        print("{0} rocks!".format(x))
    return inner
```

```
fn = outer()
```

fn.__code__.co_freevars → ('x',) (a is not a free variable)

fn.__closure__ → (<cell at 0xA500: str object at 0xFF100>,)

```
def outer():
    x = 'python'
    print(hex(id(x)))
    def inner():
        print(hex(id(x)))
        print("{0} rocks!".format(x))
    return inner
```

```
fn = outer()
fn()
```



indirect reference

Modifying free variables

```
def counter():    closure  
    count = 0  
  
    def inc():  
        nonlocal count  
        count += 1  
        return count
```

```
return inc
```

```
fn = counter()
```

`fn()` → 1 `count`'s (indirect) reference changed from the object `0` to the object `1`

`fn()` → 2

`count` is a free variable

it is bound to the cell `count`

`fn` → `inc + count` → 0

Multiple Instances of Closures

Every time we run a function, a new scope is created.

If that function generates a closure, a new closure is created every time as well

```
def counter(): closure  
    count = 0  
  
    def inc():  
        nonlocal count  
        count += 1  
        return count  
  
    return inc
```

```
f1 = counter()  
f2 = counter()  
  
f1() → 1  
f1() → 2  
f1() → 3  
  
f2() → 1
```

f1 and **f2** do not have
the same extended
scope

they are different instances of the
closure

the cells are different

Shared Extended Scopes

```
def outer( ):
```

```
    count = 0
```

```
    def inc1( ):
        nonlocal count
        count += 1
        return count
```

```
    def inc2( ):
        nonlocal count
        count += 1
        return count
```

```
    return inc1, inc2
```

```
f1, f2 = outer()
```

```
f1( ) → 1
```

```
f2( ) → 2
```

count is a free variable – bound to count in the extended scope

count is a free variable – bound to the same count

returns a tuple containing both closures

Shared Extended Scopes

You may think this shared extended scope is highly unusual... but it's not!

```
def adder(n):  
    def inner(x):  
        return x + n  
  
    return inner
```

```
add_1 = adder(1)  
add_2 = adder(2)      Three different closures – no shared scopes  
add_3 = adder(3)
```

```
add_1(10)    → 11  
add_2(10)    → 12  
add_3(10)    → 13
```

Shared Extended Scopes

But suppose we tried doing it this way:

```
adders = []
for n in range(1, 4):
    adders.append(lambda x: x + n)
```

n = 1: the free variable in the lambda is n, and it is bound to the n we created in the loop

n = 2: the free variable in the lambda is n, and it is bound to the (same) n we created in the loop

n = 3: the free variable in the lambda is n, and it is bound to the (same) n we created in the loop

Now we could call the adders in the following way:

adders[0](10) → 13

adders[1](10) → 13

adders[2](10) → 13

Remember, Python does not "evaluate" the free variable n until the adders[i] function is called

Since all three functions in adders are bound to the same n

by the time we call adders[0], the value of n is 3

(the last iteration of the loop set n to 3)

Nested Closures

```
def incrementer(n):
    # inner + n is a closure
    def inner(start):
        current = start
        # inc + current + n is a closure
        def inc():
            nonlocal current
            current += n
            return current

        return inc
    return inner
```

```
(inner)
fn = incrementer(2) → fn.__code__.co_freevars → 'n'  n=2
(inc)
inc_2 = fn(100) → inc_2.__code__.co_freevars → 'current', 'n'
(calls inc)                                     current=100, n=2
inc_2()      → 102   (current = 102, n=2)
inc_2()      → 104   (current = 104, n=2)
```

DECORATORS

PART 1

Decorators Recall the simple closure example we did which allowed us to maintain a count of how many times a function was called:

```
def counter(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print('Function {0} was called {1} times'.format(fn.__name__, count))
        return fn(*args, **kwargs)
    return inner
```

using `*args, **kwargs` means we can call any function `fn` with any combination of positional and keyword-only arguments

```
def add(a, b=0):
    return a + b
add = counter(add)
```

```
result = add(1, 2) → Function add was called 1 times
                           → result = 3
```

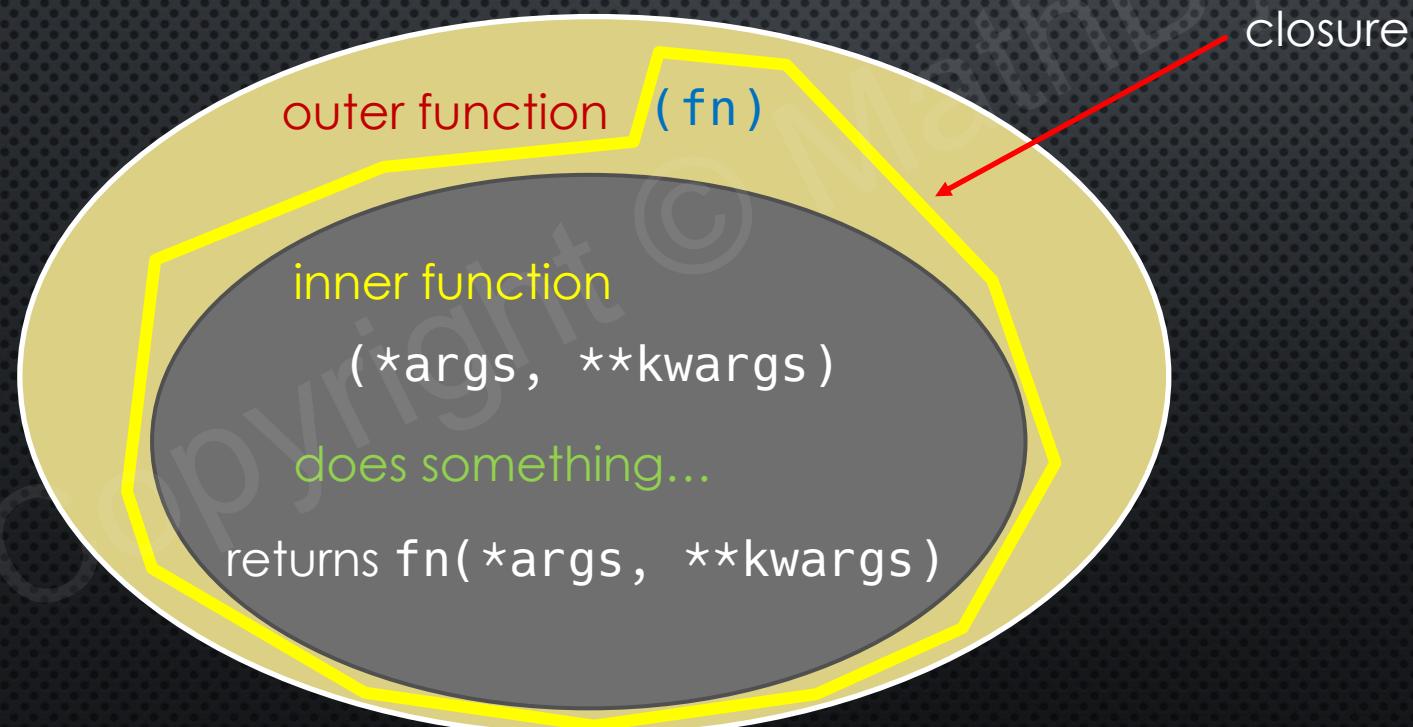
We essentially modified our `add` function by wrapping it inside another function that added some functionality to it

We also say that we **decorated** our function `add` with the function `counter`
And we call `counter` a **decorator** function

Decorators

In general a **decorator** function:

- takes a function as an argument
- returns a closure
- the closure usually accepts any combination of parameters
- runs some code in the inner function (closure)
- the closure function calls the original function using the arguments passed to the closure
- returns whatever is returned by that function call



Decorators and the @ Symbol

In our previous example, we saw that `counter` was a `decorator` and we could `decorate` our `add` function using: `add = counter(add)`

In general, if `func` is a `decorator` function, we `decorate` another function `my_func` using:

```
my_func = func(my_func)
```

This is so common that Python provides a convenient way of writing that:

```
@counter  
def add(a, b):  
    return a + b
```

is the same as writing

```
def add(a, b):  
    return a + b
```

```
add = counter(add)
```

```
@func  
def my_func(...):  
    ...
```

is the same as writing

```
def my_func(...):  
    ...
```

```
my_func = func(my_func)
```

Introspecting Decorated Functions

Let's use the same `count` decorator.

```
@counter
def mult(a, b, c=1):
    """
        returns the product of three values
    """
    return a * b * c
```

```
def counter(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print('{0} was called {1} times'.format(fn.__name__, count))
        return fn(*args, **kwargs)
    return inner
```

remember we could equally have written:
`mult = counter(mult)`

`mult.__name__`

→ `inner` not `mult`

`mult`'s name "changed" when we decorated it
they are not the same function after all

`help(mult)`

→ Help on function `inner` in module `_main_`:
`inner(*args, **kwargs)`

We have also "lost" our docstring,
and even the original function signature

Even using the `inspect` module's `signature` does not yield better results

One approach to fixing this

We could try to fix this problem, at least for the docstring and function name as follows:

```
def counter(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print('Function {} was called {} times'.format(fn.__name__, count))
        return fn(*args, **kwargs)
    inner.__name__ = fn.__name__
    inner.__doc__ = fn.__doc__
    return inner
```

But this doesn't fix losing the function signature – doing so would be quite complicated

Instead, Python provides us with a special function that we can use to fix this

The `functools.wraps` function

The `functools` module has a `wraps` function that we can use to fix the metadata of our `inner` function in our decorator

```
from functools import wraps
```

In fact, the `wraps` function is itself a decorator

but it needs to know what was our "original" function – in this case `fn`

```
def counter(fn):
    count = 0
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(count)
        return fn(*args, **kwargs)
    inner = wraps(fn)(inner)
    return inner
```

```
def counter(fn):
    count = 0
    @wraps(fn)
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(count)
        return fn(*args, **kwargs)
    return inner
```

```
def counter(fn):
    count = 0
    @wraps(fn)
    def inner(*args, **kwargs):
        nonlocal count
        count += 1
        print(count)
        return fn(*args, **kwargs)
    return inner
```

```
@counter
def mult(a:int, b:int, c:int=1):
    """
        returns the product of three values
    """
    return a * b * c
```

```
help(mult) → Help on function mult in module __main__:
mult(a:int, b:int, c:int=1)
    returns the product of three values
```

And introspection using the `inspect` module works as expected:

```
inspect.signature(mult) → <Signature (a:int, b:int, c:int=1)>
```

You don't have to use `@wraps`, but it will make debugging easier!

DECORATORS

PART 2

Decorator Parameters

In the previous videos we saw some built-in decorators that can handle some arguments:

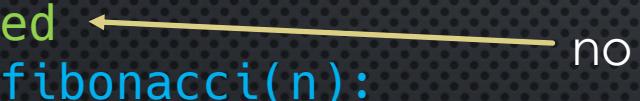
```
@wraps(fn)
def inner():
    ...
@lru_cache(maxsize=256)
def factorial(n):
    ...
```



function call

This should look quite different from the decorators we have been creating and using:

```
@timed
def fibonacci(n):
    ...
no function call
```



The `timed` decorator

```
def timed(fn):
    from time import perf_counter

    def inner(*args, **kwargs):
        total_elapsed = 0
        for i in range(10): ← hardcoded value 10
            start = perf_counter()
            result = fn(*args, **kwargs)
            total_elapsed += (perf_counter() - start)
        avg_elapsed = total_elapsed / 10
        print(avg_elapsed)
        return result
    return inner
```

`@timed`

```
def my_func():
...

```

OR

```
my_func = timed(my_func)
```

One Approach

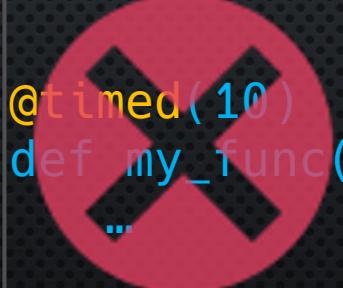
```
def timed(fn, reps):  
    from time import perf_counter  
  
    def inner(*args, **kwargs):  
        total_elapsed = 0  
        for i in range(reps):  
            start = perf_counter()  
            result = fn(*args, **kwargs)  
            total_elapsed += (perf_counter() - start)  
    avg_elapsed = total_elapsed / reps  
    print(avg_elapsed)  
    return result  
return inner
```

```
my_func = timed(my_func, 10)
```



extra parameter

free variable



```
@timed(10)  
def my_func():  
    ...
```

Rethinking the solution

```
@timed  
def my_func():  
    ...  
  
my_func = timed(my_func)
```

So, `timed` is a function that `returns` that `inner` closure that contains our original function

In order for this to work as intended:

```
@timed(10)  
def my_func():  
    ...
```

`timed(10)` will need to `return` our original `timed` decorator when `called`

```
dec = timed(10) ← timet(10) returns a decorator  
@dec  
def my_func(): ← and we decorate our function with dec  
    ...
```

Nested closures to the rescue!

our original decorator

```
def outer(reps):  
    def timed(fn):  
        from time import perf_counter  
  
        def inner(*args, **kwargs):  
            total_elapsed = 0  
            for i in range(reps):  
                start = perf_counter()  
                result = fn(*args, **kwargs)  
                total_elapsed += (perf_counter() - start)  
            avg_elapsed = total_elapsed / reps  
            print(avg_elapsed)  
            return result  
        return inner  
    return timed
```

free variable bound to `reps` in `outer`

calling `outer(n)` returns our original decorator with `reps` set to `n` (free variable)

`my_func = outer(10)(my_func)`

OR

`@outer(10)`
`def my_func():`

...

Decorator Factories

The `outer` function is not itself a decorator

instead it `returns` a `decorator` when `called`

and any arguments passed to `outer` can be referenced (as free variables) inside our decorator

We call this `outer` function a `decorator factory` function

(it is a function that `creates` a new `decorator` each time it is `called`)

And finally...

To wrap things up, we probably don't want our decorator call to look like:

```
@outer(10)
def my_func():
    ...
```

It would make more sense to write it this way:

```
@timed(10)
def my_func():
    ...
```

All we need to do is change the names of the `outer` and `timed` functions

```
def timed(reps):    ←————— this was outer
                    |
    def dec(fn):   ←————— this was timed
        from time import perf_counter
                    |
                    |
@wraps(fn)   ←————— we can still use @wraps
def inner(*args, **kwargs):
    total_elapsed = 0
    for i in range(reps):
        start = perf_counter()
        result = fn(*args, **kwargs)
        total_elapsed += (perf_counter() - start)
    avg_elapsed = total_elapsed / reps
    print(avg_elapsed)
    return result
    |
    return inner
return dec
                    |
@timed(10)
def my_func():
    ...
    ...
```

TUPLES

INTRODUCTION

Tuples are...

read-only lists... at least that's how many introductions to Python will present tuples!!

This isn't wrong, but there's a lot more going on with tuples...

If you only think of tuples as read-only lists, you're going to miss out on some interesting ideas

We really need to think of tuples also as data records position of value has meaning

This is why we are going to start looking at tuples before we even cover sequence types

We are going to focus here on tuples as a data records or structures

We will also look at named tuples

TUPLES

AS DATA STRUCTURES

Tuples vs Lists vs Strings

Tuples

containers

order matters

Heterogeneous / Homogeneous

indexable

iterable

immutable

fixed length

fixed order

cannot do in-place sorts

cannot do in-place reversals

Lists

containers

order matters

Heterogeneous / Homogeneous

indexable

iterable

mutable

length can change

order of elements can change

can do in-place sorts

can do in-place reversals

Strings

containers

order matters

Homogeneous

indexable

iterable

immutable

fixed length

fixed order

Immutability of Tuples

elements cannot be added or removed

the order of elements cannot be changed

works well for representing data structures:

Point: (10, 20)

1st element is the x-coordinate

2nd element is the y-coordinate

Circle: (0, 0, 10)

1st element is the x-coordinate of the center

2nd element is the y-coordinate of the center

3rd element is the radius

City: ('London', 'UK', 8_780_000)

1st element is the name of a city

2nd element is the country

3rd element is the population

The position of the data has meaning

Tuples as Data Records

Think of a tuple as a data record where the position of the data has meaning

```
london = ('London', 'UK', 8_780_000)  
new_york = ('New York', 'USA', 8_500_000)  
beijing = ('Beijing', 'China', 21_000_000)
```

Because tuples, strings and integers are immutable, we are guaranteed that the data and data structure for `london` will never change

We can have a list of these tuples:

```
cities = [('London', 'UK', 8_780_000),  
         ('New York', 'USA', 8_500_000),  
         ('Beijing', 'China', 21_000_000)]
```

Extracting data from Tuples

Since tuples are sequences just like strings and lists, we can retrieve items by index

```
london = ('London', 'UK', 8_780_000)
```

```
city = london[0]
```

```
country = london[1]
```

```
population = london[2]
```

```
cities = [('London', 'UK', 8_780_000),  
          ('New York', 'USA', 8_500_000),  
          ('Beijing', 'China', 21_000_000)]
```

```
total_population = 0  
for city in cities:  
    total_population += city[2]
```

You'll notice how the list of cities is **homogeneous** (contains cities only)

But a city (the tuple) is **heterogeneous**

Extracting data from Tuples

We can also use tuple **unpacking**

We actually already know how to do this – we covered this in the section on function arguments

```
new_york = ('New York', 'USA', 8_500_000)
```

packed three values into a tuple

```
city, country, population = new_york
```

unpacked tuple

```
city, country, population = ('New York', 'USA', 8_500_000)
```

```
city, country, population = 'New York', 'USA', 8_500_000
```

Dummy Variables

This is something you're likely to run across when you look at Python code that uses tuple unpacking

Sometimes, we are only interested in a subset of the data fields in a tuple, not all of them

Suppose we are interested only in the city name and the population:

```
city, _, population = ('Beijing', 'China', 21_000_000)
```

`_` is actually a legal variable name – so there's nothing special about it

but by convention, we use the underscore to indicate this is a variable we don't care about

in fact, we could just have used:

```
city, ignored, population = ('Beijing', 'China', 21_000_000)
```

Dummy Variables

It's also used in extended unpacking too

```
record = ('DJIA', 2018, 1, 19, 25987.35, 26071.72, 25942.83, 26071.72)  
symbol, year, month, day, open, high, low, close = record
```

Let's say we are only interested in the `symbol`, `year`, `month`, `day` and `close` fields

We could do it this way:

```
symbol = record[0]  
year = record[1]  
month = record[2]  
day = record[3]  
close = record[7]
```

looks really bad!



```
symbol, year, close = record[0], record[1], record[7]    awful!
```



```
symbol, year, month, day, *_, close = record
```

```
symbol, year, month, day, *ignored, close = record
```

NAMED TUPLES

Copyright © 2018

Tuple as Data Structure

We have seen how we interpreted tuples as data structures

The position of the object contained in the tuple gave it meaning

For example, we can represent a 2D coordinate as: $(10, 20)$



If `pt` is a position tuple, we can retrieve the `x` and `y` coordinates using:

$$x, y = pt \quad \text{or} \quad x = pt[0] \\ y = pt[1]$$

So, for example, to calculate the distance of `pt` from the origin we could write:

```
dist = math.sqrt(pt[0] ** 2 + pt[1] ** 2)
```

Now this is not very readable, and if someone sees this code they will have to know that `pt[0]` means the x-coordinate and `pt[1]` means the y-coordinate.

This is not very transparent.

Using a Class Instead

At this point, in order to make things clearer for the reader (not the compiler, the reader), we might want to approach this using a class instead.

<pre>class Point2D: def __init__(self, x, y): self.x = x self.y = y</pre>	<pre>pt = Point2D(10, 20) distance = sqrt(pt.x ** 2 + pt.y ** 2)</pre>	
<pre>class Stock: def __init__(self, symbol, year, month, day, open, high, low, close): self.symbol = symbol self.year = year self.month = month self.day = day self.open = open self.high = high self.low = low self.close = close</pre>	<p>Class Approach</p> <p>djia.symbol</p> <p>djia.open</p> <p>djia.close</p> <p>djia.high - djia.low</p>	<p>Tuple Approach</p> <p>djia[0]</p> <p>djia[4]</p> <p>djia[7]</p> <p>djia[5] - djia[6]</p>

Extra Stuff

At the very least we should implement the `__repr__` method

→ `Point(x=10, y=20)`

We probably should implement the `__eq__` method too

→ `Point(10, 20) == Point(10, 20) → True`

```
class Point2D:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __repr__(self):  
        return f'Point2D(x={self.x}, y={self.y})'  
  
    def __eq__(self, other):  
        if isinstance(other, Point2D):  
            return self.x == other.x and self.y == other.y  
        else:  
            return False
```

Named Tuples to the rescue

There are other reasons to seek another approach. I cover some of those in the coding video

Amongst other things, Point2D objects are **mutable** – something we may not want!

There's a lot to like using tuples to represent simple data structures

The real drawback is that we have to know what the positions mean, and remember this in our code

If we ever need to change the structure of our tuple in our code (like inserting a value that we forgot)
most likely our code will break!

```
eric = ('Idle', 42)
```

```
last_name, age = eric
```

```
eric = ('Eric', 'Idle', 42)
```

```
last_name, age = eric
```

Broken!!

Class approach:

```
last_name = eric.last_name  
age = eric.age
```

Named Tuples to the rescue

So what if we could somehow combine these two approaches, essentially creating tuples where we can, in addition, give meaningful names to the positions?

That's what `namedtuples` essentially do

They subclass `tuple`, and add a layer to assign property names to the positional elements

Located in the `collections` standard library module

```
from collections import namedtuple
```

`namedtuple` is a `function` which `generates` a new `class` → `class factory`

that new class `inherits` from `tuple`

but also provides `named properties` to access elements of the tuple

but an instance of that class `is still a tuple`

Generating Named Tuple Classes

We have to understand that `namedtuple` is a class factory

When we use it, we are essentially creating a new class, just as if we had used `class` ourselves

`namedtuple` needs a few things to generate this class:

- the class name we want to use
- a sequence of field names (strings) we want to assign, in the order of the elements in the tuple
 - field names can be any valid variable name
 - except that they cannot start with an underscore

The return value of the call to `namedtuple` will be a class

We need to assign that class to a variable name in our code so we can use it to construct instances

In general, we use the same name as the name of the class that was generated

```
Point2D = namedtuple('Point2D', ['x', 'y'])
```

Generating Named Tuple Classes

```
Point2D = namedtuple('Point2D', ['x', 'y'])
```

We can create **instances** of **Point2D** just as we would with any class (since it **is** a class)

```
pt = Point2D(10, 20)
```

The variable name that we use to assign to the class generated and returned by **namedtuple** is arbitrary

```
Pt2D = namedtuple('Point2D', ['x', 'y'])  
pt = Pt2D(10, 20)
```

Generating Named Tuple Classes

```
class MyClass:  
    pass
```

```
MyClassAlias = MyClass
```

```
instance_1 = MyClass()
```

```
instance_2 = MyClassAlias()
```



0xFF300

Class:
MyClass

instantiates the same class

Similarly

```
Pt2DAlias = namedtuple('Point2D', ['x', 'y'])
```

Variable: Pt2DAlias



0xFF900

Class:
Point2D

This is the same concept as aliasing a function, or assigning a lambda function to a variable name!

Generating Named Tuple Classes

There are many ways we can provide the list of field names to the `namedtuple` function

- a list of strings
 - a tuple of strings
 - a single string with the field names separated by whitespace or commas
- in fact any sequence, just remember that order matters!

```
namedtuple('Point2D', ['x', 'y'])
```

```
namedtuple('Point2D', ('x', 'y'))
```

```
namedtuple('Point2D', 'x, y')
```

```
namedtuple('Point2D', 'x y')
```

Instantiating Named Tuples

After we have created a named tuple class, we can instantiate them just like an ordinary class

In fact, the `__new__` method of the generated class uses the `field names` we provided as param names

```
Point2D = namedtuple('Point2D', 'x y')
```

We can use `positional arguments`:

```
pt1 = Point2D(10, 20)      10 → x    20 → y
```

And even `keyword arguments`:

```
pt2 = Point2D(x=10, y=20)  10 → x    20 → y
```

Accessing Data in a Named Tuple

Since named tuples are also regular tuples, we can still handle them just like any other tuple

- by index
- slice
- iterate

```
Point2D = namedtuple('Point2D', 'x y')
```

```
pt1 = Point2D(10, 20)           isinstance(pt1, tuple) → True
```

```
x, y = pt1
```

```
x = pt1[0]
```

```
for e in pt1:  
    print(e)
```

Accessing Data in a Named Tuple

But now, in addition, we can also access the data using the field names:

```
Point2D = namedtuple('Point2D', 'x y')
pt1 = Point2D(10, 20)
```

pt1.x → 10

pt1.y → 20

Since namedtuple generated classes inherit from tuple

pt1 is a tuple, and is therefore immutable

pt1.x = 100 will not work!

```
class Point2D(tuple):
```

...

The `rename` keyword-only argument for `namedtuple`

Remember that field names for named tuples must be valid identifiers, but cannot start with an underscore

This would **not** work: `Person = namedtuple('Person', 'name age _ssn')`



`namedtuple` has a keyword-only argument, `rename` (defaults to `False`) that will automatically rename any invalid field name

uses convention: `_ {position in list of field names}`

This **will** now work:

`Person = namedtuple('Person', 'name age _ssn', rename=True)`

And the actual field names would be:

`name` `age` `_2`

Introspection

We can easily find out the field names in a named tuple generated class

class property → `_fields`

```
Person = namedtuple('Person', 'name age _ssn', rename=True)
```

```
Person._fields → ('name', 'age', '_ssn')
```

Introspection

Remember that `namedtuple` is a class factory, i.e. it generates a class

We can actually see what the code for that class is, using the class property `_source`

```
Point2D = namedtuple('Point2D', 'x y')

Point2D._source      → lots of code omitted

class Point2D(tuple):
    'Point2D(x, y)'

    def __new__(_cls, x, y):
        'Create new instance of Point2D(x, y)'
        return _tuple.__new__(_cls, (x, y))

    def __repr__(self):
        'Return a nicely formatted representation string'
        return self.__class__.__name__ + '(x=%r, y=%r)' % self

    x = _property(_itemgetter(0), doc='Alias for field number 0')

    y = _property(_itemgetter(1), doc='Alias for field number 1')
```

Extracting Named Tuple Values to a Dictionary

Instance method: `_asdict()`

that creates a dictionary of all the named values in the tuple

```
Point2D = namedtuple('Point2D', 'x y')
pt1 = Point2D(10, 20)
```

```
pt1._asdict()      → {'x': 10, 'y': 20}
```

NAMED TUPLES

MODIFYING AND EXTENDING

Named Tuples are Immutable

So how can we "change" one or more values inside the tuple?

Just like with strings, we have to create a **new** tuple, with the modified values

```
Point2D = namedtuple('Point2D', 'x y')
pt = Point2D(0, 0)
```

Suppose we need to change the value of the x coordinate:

Simple approach: `pt = Point2D(100, pt.y)`

Note that the memory address of `pt` has now **changed**

Drawback

This simple approach can work well, but it has a major drawback

```
Stock = namedtuple('Stock', 'symbol year month day open high low close')
djia = Stock('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_393)
```

Suppose we only want to change the `close` field

```
djia = Stock(djia.symbol,
             djia.year,
             djia.month,
             djia.day,
             djia.open,
             djia.high,
             djia.low,
             26_394)
```

painful!

Maybe slicing or unpacking?

```
djia = Stock('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_393)
```

```
current = djia[:7] current → ('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260)
```

```
*current, _ = djia current → ['DJIA', 2018, 1, 25, 26_313, 26_458, 26_260]
```

```
djia = Stock(*current, 26_394)
```

We can also use the `_make` class method – but we need to create an iterable that contains all the values first:

```
new_values = current + (26_394,) new_values = current.append(26_394)
```

```
new_values → 'DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_394
```

```
djia = Stock._make(new_values)
```

iterable

This still has drawbacks

```
djia = Stock('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_393)
```

What if we wanted to change a value in the middle, say `day`?

Cannot use extended unpacking (only one starred value in extending unpacking)

```
*pre, day, *post = djia    makes no sense...
```

Slicing will work:

```
pre = djia[:3]
post = djia[4:]
```

```
new_values = pre + (26,) + post
```

```
new_values → ('DJIA', 2018, 1, 26, 26_313, 26_458, 26_260, 26_394)
```

```
djia = Stock(*new_values)
```

But even this still has drawbacks!

```
djia = Stock('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_393)
```

How about modifying both the `day` and the `high` values?

```
new_values = djia[:3] + (26,) + djia[4:5] + (26_459,) + djia[6:]
```

```
djia = Stock(*new_values)
```

This is just unreadable and extremely error prone!

There has to be a better way!

The `_replace` instance method

Named tuples have a very handy instance method, `_replace`

It will copy the named tuple into a new one, replacing any values from keyword arguments

The keyword arguments are simple the field names in the tuple and the new value

The keyword name must match an existing field name

```
Stock = namedtuple('Stock', 'symbol year month day open high low close')
```

```
djia = Stock('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_393)
```

```
djia = djia._replace(day=26, high=26_459, close=26_394)
```

```
djia → 'DJIA', 2018, 1, 26, 26_313, 26_459, 26_260, 26_394
```

Note that the memory address of `djia` has now changed

Extending a Named Tuple

Sometimes we want to create named tuple that extends another named tuple, appending one or more fields

```
Stock = namedtuple('Stock', 'symbol year month day open high low close')
```

We want to create a new named tuple class, `StockExt` that adds a single field, `previous_close`

When dealing with classes, this is sometimes done by using subclassing.

But this not easy to do with named tuples

and there's a cleaner way of doing it anyway

Extending a Named Tuple

```
Point2D = namedtuple('Point2D', 'x y')
```

Let's say we want to create a **Point3D** named tuple that has an extra parameter

Yes, the obvious, and simplest approach here is best:

```
Point3D = namedtuple('Point3D', 'x y z')
```

But what happens if you have a lot of fields in the named tuple? Code is not as clean anymore...

```
Stock = namedtuple('Stock', 'symbol year month day open high low close')
```

```
StockExt = namedtuple('Stock', 'symbol year month day open high low close previous_close')
```

How about re-using the existing field names in **Stock**?

Extending a Named Tuple

```
Stock = namedtuple('Stock', 'symbol year month day open high low close')
```

```
Stock._fields → 'symbol', 'year', 'month', 'day', 'open', 'high', 'low', 'close'
```

We can then create a new named tuple by "extending" the `_fields` tuple

```
new_fields = Stock._fields + ('previous_close', )
```

```
StockExt = namedtuple('StockExt', new_fields)
```

Extending a Named Tuple

We can also easily use an existing `Stock` instance to create a new `StockExt` instance with the same common values, adding in our new `previous_close` value:

```
Stock = namedtuple('Stock', 'symbol year month day open high low close')
```

```
StockExt = namedtuple('StockExt', Stock._fields + ('previous_close', ))
```

```
djia = Stock('DJIA', 2018, 1, 25, 26_313, 26_458, 26_260, 26_393)
```

```
djia_ext = StockExt(*djia, 26_000)
```

or

```
djia_ext = StockExt._make(djia + (26_000, ))
```

NAMED TUPLES

DOCSTRINGS AND DEFAULT VALUES

Default Docs for Named Tuples

When we create a named tuple class, default docstrings are created

```
Point2D = namedtuple('Point2D', 'x y')
Point2D.__doc__      → Point2D(x, y)
Point2D.x.__doc__    → Alias for field number 0
Point2D.y.__doc__    → Alias for field number 1
```

```
help(Point2D)   →  class Point2D(builtins.tuple)
                           Point2D(x, y)

                           x
                           Alias for field number 0

                           y
                           Alias for field number 1
```

Overriding DocStrings

We can override the docstrings simply by specifying values for the `__doc__` properties
(this is not unique to named tuples!)

```
Point2D.__doc__ = 'Represents a 2D Cartesian coordinate.'
```

```
Point2D.x.__doc__ = 'x coordinate'
```

```
Point2D.y.__doc__ = 'y coordinate'
```

```
help(Point2D)    → class Point2D(builtins.tuple)
                           Represents a 2D Cartesian coordinate.

                           x
                           x coordinate

                           y
                           y coordinate
```

Default Values

The `namedtuple` function does not provide us a way to define default values for each field

Two approaches to this:

Using a Prototype

Create an instance of the named tuple with **default** values - the **prototype**

Create any additional instances of the named tuple using the `prototype._replace` method

You will need to supply a default for every field (can be `None`)

Using the `__defaults__` property

Directly set the defaults of the named tuple constructor (the `__new__` method)

You do not need to specify a default for every field

Remember that you cannot have non-defaulted parameters
after the first defaulted parameter

`def func(a, b=10, c=20)` 

`def func(a, b=10, c)`  

Using a Prototype

```
Vector2D = namedtuple('Vector2D', 'x1 y1 x2 y2 origin_x origin_y')  
  
vector_zero = Vector2D(x1=0, y1=0, x2=0, y2=0, origin_x=0, origin_y=0)
```

or

```
vector_zero = Vector2D(0, 0, 0, 0, 0, 0)
```

`vector_zero → Vector2D(x1=0, y1=0, x2=0, y2=0, origin_x=0, origin_y=0)`

To construct a new instance of `Vector2D` we now use `vector_zero._replace` instead:

```
v1 = vector_zero._replace(x1=10, y1=10, x2=20, y2=20)
```

`v1 → Vector2D(x1=10, y1=10, x2=20, y2=20, origin_x=0, origin_y=0)`

Using `__defaults__`

```
def func(a, b=10, c=20):  
    pass
```

`func.__defaults__` → (10, 20)

a	b	c
10	20	

↑
no default

The `__defaults__` property is **writable**

So we can set it to a **tuple** of our choice

Just don't provide more defaults than parameters! (extras are ignored)

Using `__defaults__`

We need to provide defaults to the constructor of our named tuple class `__new__`

```
Vector2D = namedtuple('Vector2D', 'x1 y1 x2 y2 origin_x origin_y')
```

```
Vector2D.__new__.__defaults__ = (0, 0)      x1 y1 x2 y2 origin_x origin_y  
                                         0       0
```

```
v1 = Vector2D(10, 10, 20, 20)
```

```
v1 → Vector2D(x1=10, y1=10, x2=20, y2=20, origin_x=0, origin_y=0)
```

Isn't this cleaner than the prototype approach?!!

```
v1 = vector_zero._replace(x1=10, y1=10, x2=20, y2=20)
```

MODULES
PACKAGES
PACKAGE NAMESPACES

copyrig^{ht} 2014

Modules

What are modules exactly?

→ objects of type `ModuleType`

How does Python load modules?

How to import without the import statement

Reloading modules

→ why we should not do it!

Import variants

→ `import`
`from ... import ...`
`from ... import *`

Misconceptions

`__main__`

in modules

as file names

Zip Archives

importing from a zip archive

zipping an entire Python app

creating an executable Python app in `bash`

Packages

What is a package?

Why use them?

How is it different from a module?

The role of `__init__.py` files in packages

Implicit Namespace Packages (Python 3.3+)

What are they?

How do we create and use them?

vs standard packages

IMPORT VARIANTS

AND SOME MISCONCEPTIONS

Import variants

```
# module1.py  
import math
```

is `math` in `sys.modules`?

if not, load it and insert ref

`sys.modules`

```
math    <module object>
```

add symbol `math` to `module1`'s global namespace referencing the same object

`module1.globals()`

```
math    <module object>
```

`math` symbol in namespace

(if `math` symbol already exists in `module1`'s namespace, replace reference)

Import variants

```
# module1.py  
import math as r_math
```

is `math` in `sys.modules`?

if not, load it and insert ref

add symbol `r_math` to `module1`'s global namespace referencing the same object

```
module1.globals()
```

```
r_math <module object>
```

`r_math` symbol in namespace
`math` symbol **not** in namespace

(if `r_math` symbol already exists in `module1`'s namespace, replace reference)

Import variants

```
# module1.py  
from math import sqrt
```

is `math` in `sys.modules`?

if not, load it and insert ref

`sys.modules`

```
math    <module object>
```

add symbol `sqrt` to `module1`'s global namespace referencing `math.sqrt`

`module1.globals()`

```
sqrt    <math.sqrt object>
```

`math` symbol **not** in namespace

(if `sqrt` symbol already exists in `module1`'s namespace, replace reference)

Import variants

```
# module1.py  
from math import sqrt as r_sqrt
```

is `math` in `sys.modules`?

if not, load it and insert ref

`sys.modules`

```
math <module object>
```

add symbol `r_sqrt` to `module1`'s global namespace referencing `math.sqrt`

`module1.globals()`

```
r_sqrt <math.sqrt object>
```

`math` symbol **not** in namespace

(if `r_sqrt` symbol already exists in `module1`'s namespace, replace reference)

Import variants

```
# module1.py  
from math import *
```

is `math` in `sys.modules`?

if not, load it and insert ref

`sys.modules`

```
math    <module object>
```

add "all" symbols defined in `math` to `module1`'s global namespace

`module1.globals()`

```
pi    <math.pi object>  
sin   <math.sin object>
```

and many more...

what "all" means can be defined by the module being imported

`math` symbol **not** in namespace

(if any `symbols` already exists in `module1`'s namespace, replace their reference)

Commonality

In **every** case the `math` module was loaded into memory and referenced in `sys.modules`

Running

```
from math import sqrt
```

did not "partially" load math

it only affected **what** symbols were placed in `module1`'s namespace!

Things may be different with packages, but for simple modules this is the behavior

Why `from <module> import *` can lead to bugs

```
# module1.py
from cmath import *
```

```
module1.globals()
    sqrt  <cmath.sqrt>
    ...
    ...
```

```
from math import *
```

```
module1.globals()
    sqrt  <math.sqrt>
    ...
    ...
```

Efficiency

What's more efficient?

```
import math  
or   from math import sqrt
```

importing → same amount of work

calling

```
math.sqrt(2)  
sqrt(2)
```

This first needs to find the `sqrt` symbol in `math`'s namespace

`dict` lookup → super fast!

MODULES

A QUICK RECAP

What we've seen so far...

Modules can be imported using

the `import` statement

`importlib.import_module`

When a module is imported:

system cache is checked first `sys.modules` → if in cache, just returns cached reference
otherwise:

module has to be located (found) somewhere finders e.g. `sys.meta_path`

module code has to be retrieved (loaded) loaders returned by finder → `ModuleSpec`

"empty" module typed object is created

a reference to the module is added to the system cache `sys.modules`

module is compiled

module is executed → sets up the module's namespace (`module.__dict__ is module.globals()`)

Module Finders

<code>sys.meta_path</code> →	<code>_frozen_importlib.BuiltinImporter</code>	finds built-ins, such as <code>math</code>
	<code>_frozen_importlib.FrozenImporter</code>	finds frozen modules
	<code>_frozen_importlib_external.PathFinder</code>	file-based modules

PathFinder

Finds file-based modules based on `sys.path` and package `__path__`

`sys.path` → `['/home/fmb/my-app', '/usr/lib/python36.zip', '/usr/lib/python3.6', '/usr/lib/python3.6/lib-dynload', '/usr/local/lib/python3.6/dist-packages', '/usr/lib/python3/dist-packages']`

`collections.__path__` → `['/usr/lib/python3.6/collections']`

Module Properties

built-in `import math`

`type(math)` → `module`

`math.__spec__` → `ModuleSpec(name='math',
loader=<class '_frozen_importlib.BuiltinImporter'>
origin='built-in')`

`math.__name__` → `math`

`math.__package__` → `''`

`__file__` is not an attribute of `math` (built-ins only)

Module Properties

standard library `import fractions`

`type(fractions)` → `module`

`fractions.__spec__` → `ModuleSpec(name='fractions',
loader=<_frozen_importlib_external.SourceFileLoader
object at 0x7fa9bf7ff6d8>,
origin='/usr/lib/python3.6/fractions.py')`

`fractions.__name__` → `'fractions'`

`fractions.__package__` → `''`

`fractions.__file__` → `/usr/lib/python3.6/fractions.py`

Note that `fractions.__file__` was found by `PathFinder` in one of the paths listed in `sys.path`

Module Properties

custom module `import module1`

`type(module1)` → `module`

`module1.__spec__` → `ModuleSpec(name='module1',
 loader=<_frozen_importlib_external.SourceFileLoader
 object at 0x7fd9f4c4ae48>,
 origin='/home/fmb/my-app/module1.py')`

`module1.__name__` → `module1`

`module1.__package__` → `''`

`module1.__file__` → `/home/fmb/my-app/module1.py`

Note that `module1.__file__` was found by `PathFinder` in one of the paths listed in `sys.path`

Some Notes

Python modules may reside

in the built-ins

in files on disk

they can even be pre-compiled, frozen, or even inside zip archives

anywhere else that can be accessed by a finder and a loader

custom finders/loaders → database, http, etc

Python docs:

<https://docs.python.org/3/tutorial/modules.html>

<https://docs.python.org/3/reference/import.html>

PEP 302

For file based modules ([PathFinder](#)):

They must exist in a path specified in

`sys.path`

or in a path specified by `<package>.__path__`

WHAT ARE PACKAGES?

Copyright © 2018

Packages are Modules

Packages **are** modules (but modules are not necessarily packages)

They can **contain**

modules

packages

(called **sub-packages**)

If a module is a package, it must have a value set for **`__path__`**

After you have imported a module, you can easily see if that module is a package by inspecting the **`__path__`** attribute (empty → module, non-empty → package)

Packages and File Systems

Remember that modules do not have to be entities in a file system (loaders, finders)

By the same token, packages do not have to be entities in the file system

Typically they are - just as typically modules are file system entities

But packages represent a **hierarchy** of modules / packages

`pack1.mod1`

`pack1.pack1_1.mod1_1`

dotted notation indicates the **path** hierarchy of modules / packages

and is usually found in `__path__`

Importing Nested Packages

If you have a statement in your top-level program such as:

```
import pack1.pack1_1.module1
```

The import system will perform these steps:

```
imports pack1
```

```
imports pack1.pack1_1
```

```
imports pack1.pack1_1.module1
```

The `sys.modules` cache will contain entries for:

```
pack1
```

```
pack1.pack1_1
```

```
pack1.pack1_1.module1
```

The namespace where the import was run contains:

```
pack1
```

File System Based Packages

Although modules and packages can be far more generic than file system based entities, it gets complicated!

If you're interested in this, then the first document you should read is [PEP302](#)

In this course we're going to stick to traditional [file based](#) modules and packages

File Based Packages

→ package paths are created by using file system directories and files

Remember: a package is simply a module that can contain other modules/packages

On a file system we therefore have to use directories for packages

The directory name becomes the package name

So where does the code go for the package (since it is a module)?

`__init__.py`

`__init__.py`

To define a package in our file system, we must:

- create a directory whose name will be the package name

- create a file called `__init__.py` inside that directory

That `__init__.py` file is what tells Python that the directory is a package as opposed to a standard directory

(if we don't have an `__init__.py` file, then Python creates an implicit namespace package)

- we'll discuss that later

What happens when a file based package is imported?

```
app/  
  pack1/  
    __init__.py  
    module1.py  
    module2.py
```

```
import pack1
```

the code for `pack1` is in `__init__.py`

that code is loaded, executed and cached in `sys.modules` with a key of `pack1`

it's just a module!

the symbol `pack1` is added to our namespace referencing the same object

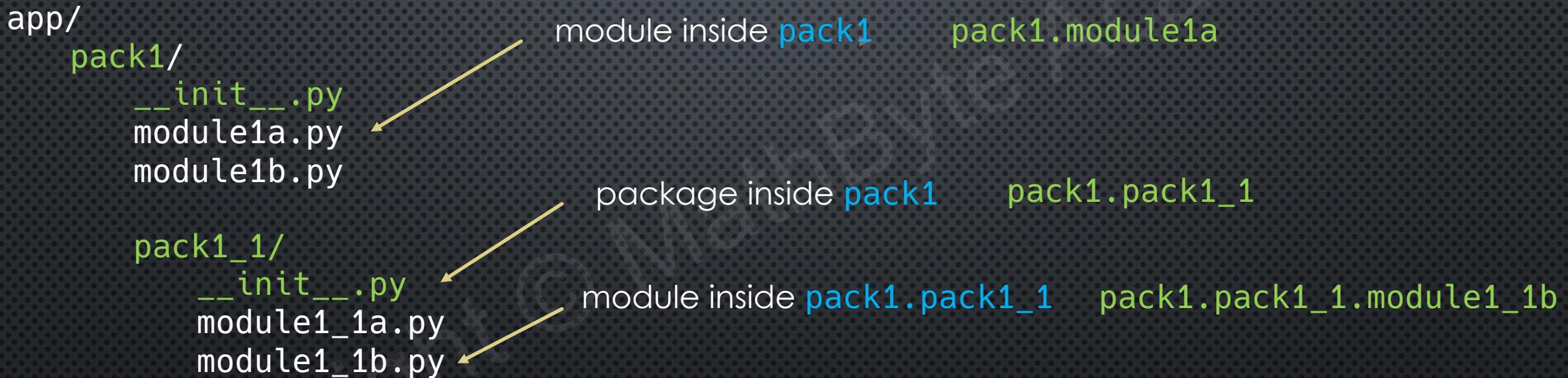
it's just a module!

but, it has a `__path__` property → file system directory path (absolute)

also has a `__file__` property → file system path to `__init__.py` (absolute)

Nested Packages

Packages can contain modules as well as packages



`__file__`, `__path__` and `__package__` Properties

Modules have `__file__` and `__package__` properties

`__file__` is the location of module code in the file system

`__package__` is the package the module code is located in

(an empty string if the module is located in the application root)

If the module is also a package, then it also has a `__path__` property

`__path__` is the location of the package (directory) in the file system

```
app/  
  module.py
```

```
pack1/  
  __init__.py  
  module1a.py  
  module1b.py
```

```
module.__file__ → .../app/module.py  
module.__path__ → not set  
module.__package__ → ''
```

```
pack1_1/  
  __init__.py  
  module1_1a.py  
  module1_1b.py
```

```
pack1.__file__ → .../app/pack1/__init__.py  
pack1.__path__ → .../app/pack1  
pack1.__package__ → pack1
```

```
pack1.module1a.__file__ → .../app/pack1/module1a.py  
pack1.module1a.__path__ → not set  
pack1.module1a.__package__ → pack1
```

```
app/  
  module.py
```

```
  pack1/  
    __init__.py  
    module1a.py  
    module1b.py
```

```
  pack1_1/  
    __init__.py  
    module1_1a.py  
    module1_1b.py
```

pack1.pack1_1.__file__ → .../app/pack1/pack1_1/__init__.py

pack1.pack1_1.__path__ → .../app/pack1/pack1_1

pack1.pack1_1.__package__ → pack1.pack1_1

pack1.pack1_1.module1_1a.__file__ → .../app/pack1/pack1_1/module1_1a.py

pack1.pack1_1.module1_1a.__path__ → not set

pack1.pack1_1.module1_1a.__package__ → pack1.pack1_1

What gets loaded during the import phase?

app/

 module.py

 pack1/

 __init__.py

 module1a.py

 module1b.py

 pack1_1/

 __init__.py

 module1_1a.py

 module1_1b.py

import pack1.pack1_1.module1_1a

at the very least:

pack1 is imported and added to `sys.modules`

pack1_1 is imported and added to `sys.modules`

module1_1a is imported and added to `sys.modules`

but, modules can import other modules!

`pack1.__init__.py` could import other modules/packages

`pack1_1.__init__.py` could import other modules/packages

`module1_1a.__init__.py` could import other modules/packages

For example...

```
app/  
  module.py
```

```
pack1/  
  __init__.py  
  module1a.py  
  module1b.py
```

```
pack1_1/  
  __init__.py  
  module1_1a.py  
  module1_1b.py
```

```
# pack1.__init__.py  
  
import pack1.module1a  
import pack1.module1b  
  
import pack1.pack1_1.module1_1a
```

Just as before:

`pack1` is imported and added to `sys.modules`

`pack1_1` is imported and added to `sys.modules`

`module1_1a` is imported and added to `sys.modules`

but now also:

`module1a` is imported and added to `sys.modules`

`module1b` is imported and added to `sys.modules`

For example...

```
app/  
  module.py
```

```
pack1/  
  __init__.py  
  module1a.py  
  module1b.py
```

```
pack1_1/  
  __init__.py  
  module1_1a.py  
  module1_1b.py
```

```
# pack1.__init__.py  
  
import pack1.module1a  
import pack1.module1b  
  
import pack1
```

pack1 is imported and added to `sys.modules`
module1a is imported and added to `sys.modules`
module1b is imported and added to `sys.modules`

WHY PACKAGES?

Copyright © 2014

Code Organization, Ease of Use...

Suppose you have 50 different functions and classes in your program

api.py

connect
execute_no_result
execute_single_row
execute_multi_row

normalize_string
convert_str_to_bool
format_iso_date
current_time_utc

authenticate
validate_token
get_permissions
authorize_endpoint

User
UserProfile
Users

BlogPost
BlogPosts

RouteTable
Configuration

JSONEncoder

UnitTests

(single file)

audit_endpoint
Logger

validate_email
validate_phone
validate_name

etc...

in **one** file???



Start with Modules...

```
api/  
    api.py  
  
dbutilities.py  
  
jsonutilities.py  
  
typeconversions.py  
  
validations.py  
  
authentication.py  
  
authorization.py  
  
users.py  
  
blogposts.py  
  
logging.py  
  
unittests.py
```

better...

but still unwieldy – everything is at the top level

too many imports:

```
import dbutilities  
import jsonutilities  
import typeconversions  
import validations  
import authentication  
import authorization  
import users  
etc...
```

certain modules could be broken down further:

dbutilities → connections, queries
users → User, Users, UserProfile

certain modules belong "together":

authentication, authorization → security

So, Packages...

```
api/  
    api.py  
  
dbutilities.py  
  
jsonutilities.py  
  
typeconversions.py  
  
validations.py  
  
authentication.py  
  
authorization.py  
  
users.py  
  
blogposts.py  
  
logging.py  
  
unittests.py
```

```
api/  
    utilities/  
        __init__.py  
    database/  
        __init__.py  
        connections.py  
        queries.py  
    json/  
        __init__.py  
        encoders.py  
        decoders.py  
    security/  
        __init__.py  
        authentication.py  
        authorization.py  
    models/  
        __init__.py  
        users/  
            __init__.py  
            user.py  
            userprofile.py
```

Another Use Case

You have a module that implements 2 functions/classes for users of the module

Those two objects require 20 different helper functions and 2 additional helper classes

From module developer's perspective:

much easier to break the code down into multiple modules

From module user's perspective:

they just want a single import for the function and the class

i.e. it should look like a single module

Module Developer's Perspective



Smaller code modules, with a specific purpose, are easier to write, debug, test, and understand

Module User's Perspective

```
mylib/  
    __init__.py  
    submod1.py  
    submod2.py  
    subpack1
```

```
        __init__.py  
        pack1mod1.py  
        pack1mod2.py
```

User should not have to write:

function to be exported to user lives here

class to be exported to user lives here

```
from mylib.submod1 import my_func  
from mylib.subpack1.pack1mod2 import MyClass
```

Much easier for user if they could write:

or, simply

```
from mylib import my_func, MyClass  
  
import mylib  
mylib.my_func()  mylib.MyClass()
```

Using `__init__.py`

We can use packages' `__init__.py` code to **export** (expose) just what's needed by our users

Example:

```
mylib/
    __init__.py
    submod1.py
    submod2.py
    subpack1
        __init__.py
        pack1mod1.py
    pack1mod2.py
```

function to be exported
to user lives here

class to be exported
to user lives here

```
# mylib.__init__.py
from mylib.submod1 import my_func
from mylib.subpack1.pack1mod2 import MyClass
```

User uses it this way:

```
import mylib
mylib.my_func()
mylib.MyClass()
```

our internal implementation is "hidden"

We'll cover this in the next video

So, why Packages?

ability to break code up into smaller chunks, makes **our** code:

- easier to write
- easier to test and debug
- easier to read/understand
- easier to document

just like books are broken down into chapters, sections, paragraphs, etc.

but they can still be "stitched" together

hides inner implementation from users

makes **their** code

- easier to write
- easier to test and debug
- easier to read/understand

NAMESPACE PACKAGES

Copyright © 2014

What are Implicit Namespace Packages?

Namespace packages are package-like

directories

may contain modules

may contain nested regular packages

may contain nested namespace packages

but cannot contain `__init__.py`

These directories are implicitly made into these special types of packages

PEP 420

Mechanics

utils/	utils/ does not contains <code>__init__.py</code>	→ namespace package
validators/	validators/ does not contain <code>__init__.py</code>	→ namespace package
boolean.py	boolean.py is a file with a <code>.py</code> extension	→ module
date.py		
json/	json/ contains <code>__init__.py</code>	→ regular package
<code>__init__.py</code>		
serializers.py	serializers.py is a file with a <code>.py</code> extension	→ module
validators.py		

Regular vs Namespace Packages

Regular Package

`type` → `module`

`__init__.py` → yes

`__file__` → package `__init__`

`paths` → breaks if parent
directories change
and absolute imports
are used

single package lives in single
directory

Namespace Package

`type` → `module`

`__init__.py` → no

`__file__` → not set

`paths` → dynamic path computation
so OK if parent directories change

(your import statements will still
need to be modified)

single package can live in **multiple** (non-nested)
directories

in fact, parts of the namespace may even be
in a zip file

Example

	namespace package	regular package	app/ utils/ validators/ boolean.py
type	<u>utils</u>	<u>common</u>	common/ __init__.py
__name__	utils	module	validators/ boolean.py
__repr__()	<module utils (namespace)>	<module common from '.../app/common'>	
__path__	_Namespace(['.../app/utils'])	['.../app/utils']	
__file__	not set	.../app/common/__init__.py	
__package__	utils	common	
→ validators	utils.validators	common.validators	

Import Examples

```
utils/  
  validators/  
    boolean.py  
  date.py  
  json/  
    __init__.py  
    serializers.py  
    validators.py
```

```
import utils.validators.boolean  
from utils.validators import date  
import utils.validators.json.serializers
```

First familiarize yourself with regular packages.

Once you are completely comfortable with them, check out namespace packages if you want

Read PEP 420 – that should definitely be your starting point

EXTRAS

Copyright © 2014

What's in this section?

Tips and tricks

Pythonic code

Opiniated

Things I find interesting

Will grow over time

This is NOT going to be discussions of 3rd party Library XYZ

(As of today PyPi has **128,291** packages!)

Send me your suggestions!

ADDITIONAL RESOURCES

Copyright © 2014 by Pearson Education, Inc.

The Python documentation

That should be your top bookmark for Python

<https://docs.python.org>

Don't forget to make sure you are looking at your version of Python.

3.6 or above please!

Python » English ▾ 3.6.4 ▾ Documentation »

Quick search Go | modules | ir

Download
Download these documents

Docs for other versions
Python 3.7 (in development)
Python 3.5 (stable)
Python 2.7 (stable)
Old versions

Other resources
PEP Index
Beginner's Guide
Book List
Audio/Visual Talks

Python 3.6.4 documentation

Welcome! This is the documentation for Python 3.6.4.

Parts of the documentation:

- [What's new in Python 3.6?](#)
or all "What's new" documents since 2.0
- [Tutorial](#)
start here
- [Library Reference](#)
keep this under your pillow
- [Language Reference](#)
describes syntax and language elements
- [Python Setup and Usage](#)
how to use Python on different platforms
- [Python HOWTOs](#)
in-depth documents on specific topics
- [Indices and tables](#)

Installing Python Modules
installing from the Python Package Index & other sources

Distributing Python Modules
publishing modules for installation by others

Extending and Embedding
tutorial for C/C++ programmers

Python/C API
reference for C/C++ programmers

FAQs
frequently asked questions (with answers!)

PEP – Python Enhancement Proposals

These are a fantastic resource to understand how certain things work in Python, and why they were implemented in a certain way..

Not all PEPs actually make it into Python. Some are rejected, deferred or even withdrawn.

Reading the PEPs that have not been accepted also provides a lot of insight! A lot of thought by many people go into these PEPs, whether they make it or not.

PEP 274 -- Dict Comprehensions	
PEP:	274
Title:	Dict Comprehensions
Author:	Barry Warsaw <barry at python.org>
Status:	Final
Type:	Standards Track
Created:	25-Oct-2001
Python-Version:	2.7, 3.0 (originally 2.3)
Post-History:	29-Oct-2001

Some PEPs are for language features
some are informational only

Index page

<https://www.python.org/dev/peps/>

search on that page

But sometimes a web search such as: [Python PEP Style Guide](#) is more practical

PEP – Some Notable Ones

PEP 8 – Style Guide and Idiomatic Python

PEP 20 – Zen of Python

or just type `import this` in a Python console/Jupyter

PEP 484 – Type Hints

PEP 468 – Python 3.6 Release Schedule

PEP 537 – Python 3.7 Release Schedule

or whatever release you're interested in at the time
they provide links to other PEPs relevant to the release

And many many more, depending on what topic you're interested in

Great resource for explanations of general computer science concepts

Hash table

From Wikipedia, the free encyclopedia

Not to be confused with Hash list or Hash tree.
"Rehash" redirects here. For the South Park episode, see Rehash (South Park). For the IRC command, see List of Internet Relay Chat commands § REHASH.

In computing, a **hash table** (**hash map**) is a **data structure** which implements an **associative array abstract data type**, a structure that can map keys to values. A hash table uses a **hash function** to compute an **index** into an array of **buckets** or **slots**, from which the desired value can be found.

Ideally, the hash function will assign each key to a unique bucket, but most hash table designs employ an imperfect hash function, which might cause hash **collisions** where the hash function generates the same index for more than one key. Such collisions must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of **instructions**) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at (amortized^[2]) constant average cost per operation.^{[3][4]}

In many situations, hash tables turn out to be more efficient than search trees or any other **table** lookup structure. For this reason, they are widely used in many kinds of computer **software**, particularly for associative arrays, **database indexing**, **caches**, and **sets**.

Contents [hide]

- 1 Hashing
 - 1.1 Choosing a hash function
 - 1.2 Perfect hash function
- 2 Key statistics
- 3 Collision resolution
 - 3.1 Separate chaining
 - 3.1.1 Separate chaining with linked lists
 - 3.1.2 Separate chaining with list head cells
 - 3.1.3 Separate chaining with other structures
 - 3.2 Open addressing
 - 3.2.1 Coalesced hashing
 - 3.2.2 Cuckoo hashing
 - 3.2.3 Hopscotch hashing
 - 3.3 Robin Hood hashing
- 4 See also

Hash table

Type	Unordered associative array	
Invented	1953	
Time complexity in big O notation		
Algorithm	Average	Worst case
Space	$O(n)$ ^[1]	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

hash

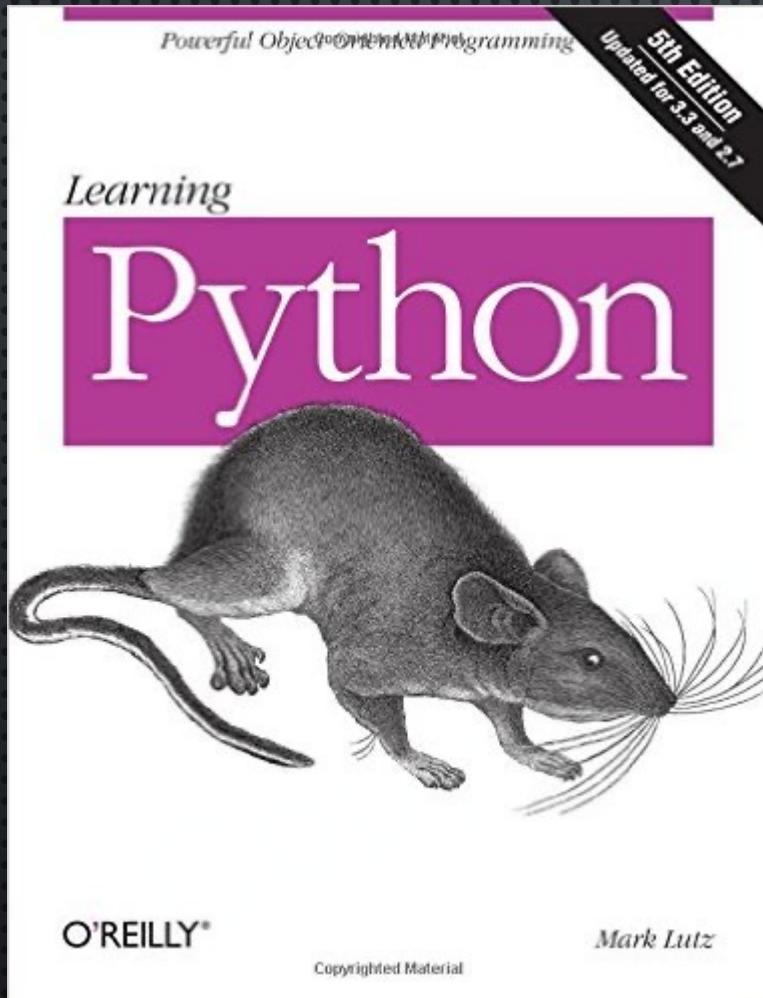
keys	function	buckets
John Smith	01	521-8976
Lisa Smith	02	521-1234
Sandra Dee	13	521-9655
	14	
	15	

A small phone book as a hash table

Books

These are my favorite Python specific go to books

not in any particular order of importance!



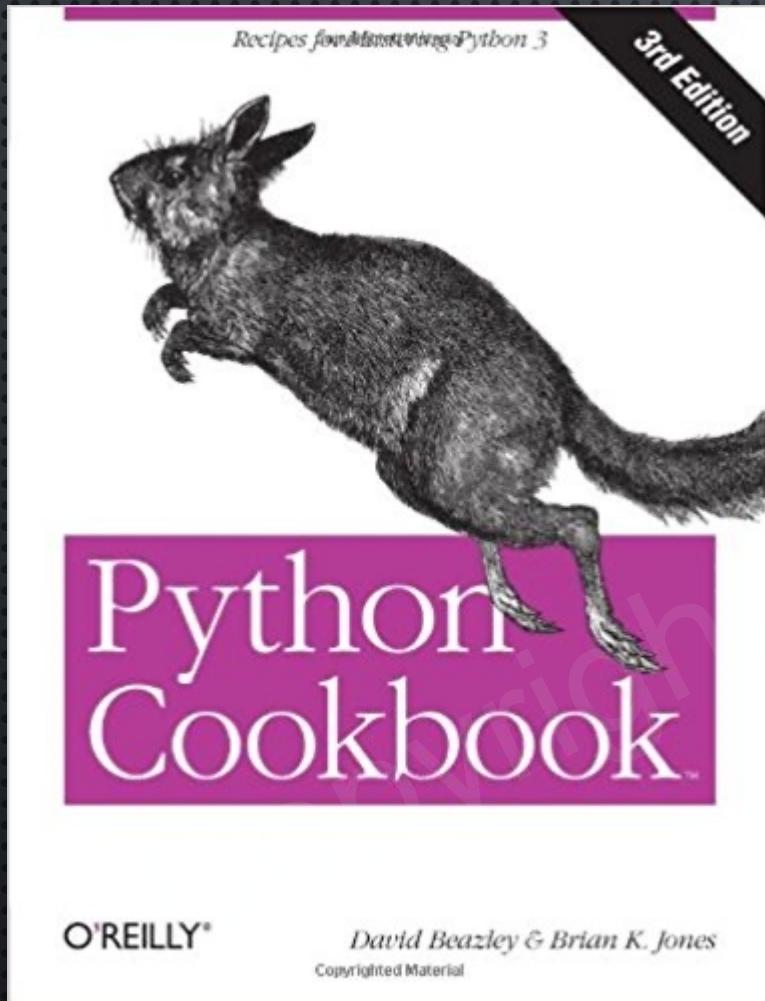
Learning Python
Mark Lutz

Books



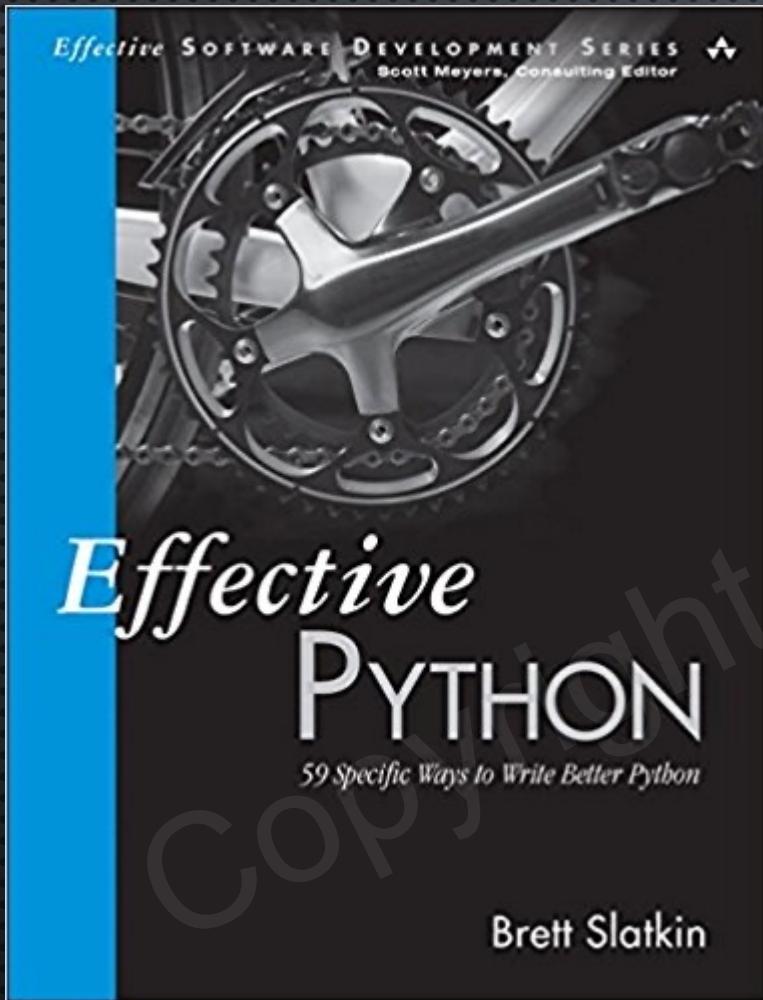
Fluent Python
Luciano Ramalho

Books



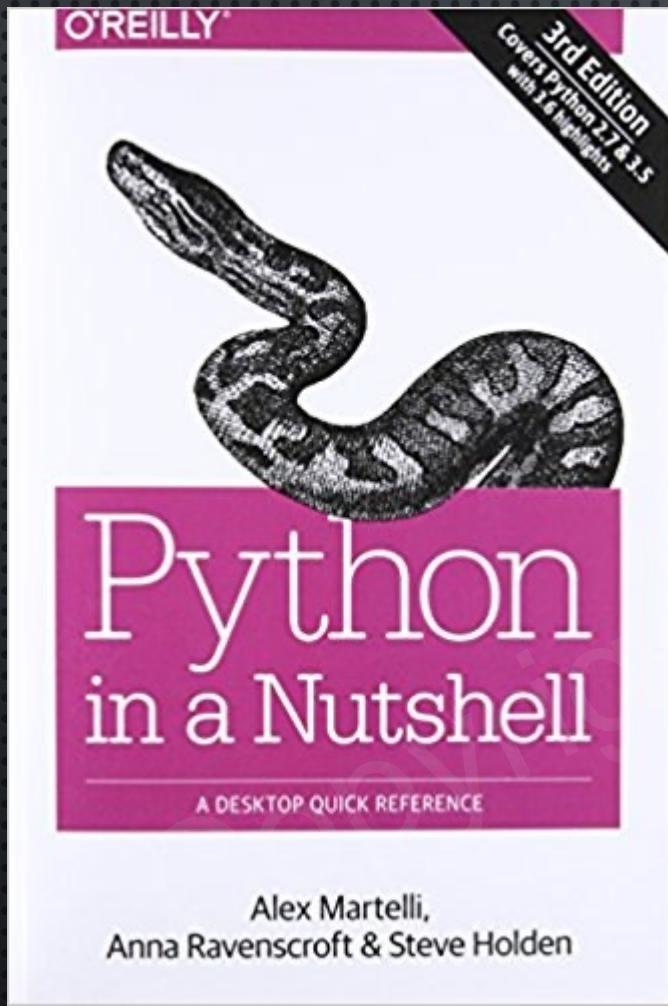
Python Cookbook
David Beazley & Brian K. Jones

Books



Effective Python: 59 Specific Ways to Write Better Python
Brett Slatkin

Books



Python in a Nutshell
Alex Martelli, Anna Ravenscroft & Steve Holden

Other Online Resources I Regularly Use

Raymond Hettinger's Twitter Feed

@raymondh

just awesome!

example:

```
#python tip: zip() with star-arguments is  
great for transposing 2-D data:  
m = [(1, 2, 3), (4, 5, 6)]  
list(zip(*m))  
[(1, 4), (2, 5), (3, 6)]
```

YouTube

Lots of great videos on Python.
Look out for PyCon videos – these are fantastic!
Anything by GvR, Raymond Hettinger, Alex Martelli...
And many more, including any library you're interested in

Planet Python Blog

<http://planetpython.org/>

Google Searches!

Stack Overflow

<https://stackoverflow.com/>