PYTHON CLASSES and INHERITANCE

(download slides and .py files from Stellar to follow along!)

6.0001 LECTURE 8

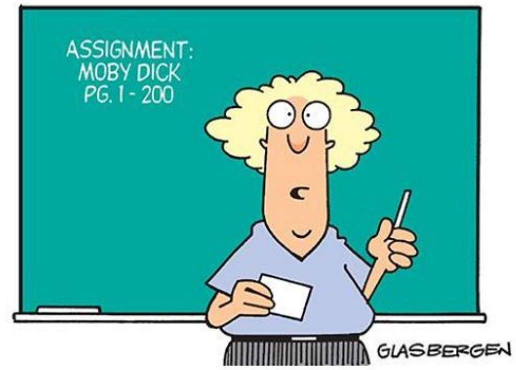
LAST TIME

- Abstract data types using classes
- `Coordinate` example
- `Fraction` example

TODAY

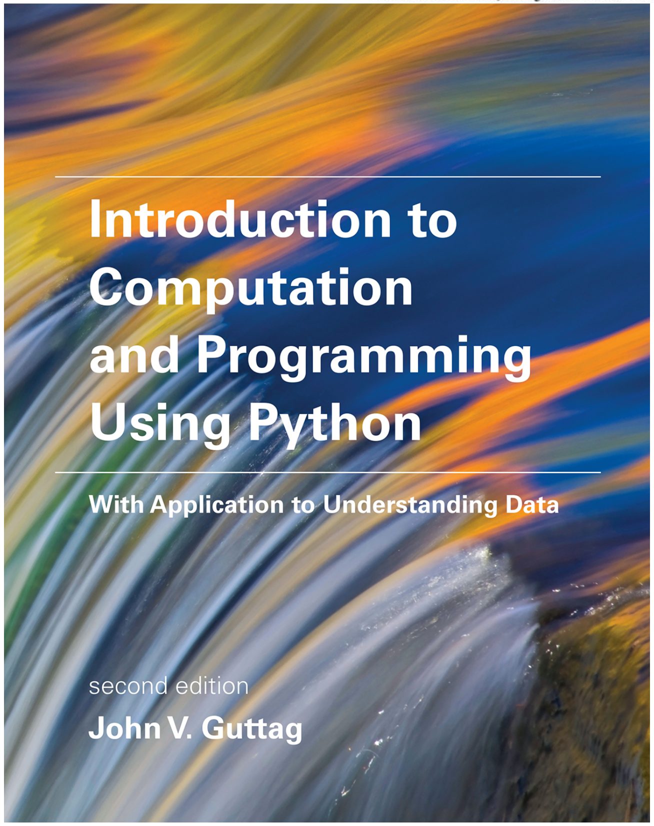
- Review classes
- More details on classes, class variables
- Inheritance and hierarchies of classes
- Introduction to algorithmic complexity

Assigned Reading



"I don't like to give a lot of homework over the weekend, so just read every other word."

- Today
  - 8.2
  - 9.1 – 9.2
- Next lecture
  - 9.3



https://mitpress.mit.edu/sites/default/files/Guttag_errata_revised_083117.pdf

THE POWER OF OBJECT ORIENTED PROGRAMMING

- **Bundle together objects** that share
  - common attributes and
  - procedures that operate on those attributes
- Use **abstraction** to make a distinction between how to implement an object versus how to use an object
- Build **layers** of object abstractions that inherit behaviors from other classes of objects
- Create our **own classes of objects** on top of Python's basic classes

Another instance of a virtuous cycle – just as defining procedures lets us create new procedures and treat as if built-in, we can create classes and treat as if built in to Python

IMPLEMENTING THE CLASS vs USING THE CLASS

- Write code from two different perspectives

Implementing a new object type with a class
- **Define** the class
- Define **data attributes** (WHAT IS the object)
- Define **methods** (HOW TO use the object)

Using the new object type in code
- Create **instances** of the object type
- Do **operations** with them

Class captures common properties and behaviors	Instances have specific values for attributes

CLASS DEFINITION OF AN OBJECT TYPE vs INSTANCE OF A CLASS

- Class name is the **type**

  `class Coordinate(object)`

- Class is defined generically
  - Use `self` to refer to some instance while defining class

  `(self.x – self.y)**2`

  - `self` is a parameter to methods in class definition

- Class defines data and methods **common across all instances**

- Instance is **one specific object**

  `coord = Coordinate(1,2)`

- Data attribute values vary between instances

  `c1 = Coordinate(1,2)`

  `c2 = Coordinate(3,4)`

  - `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects

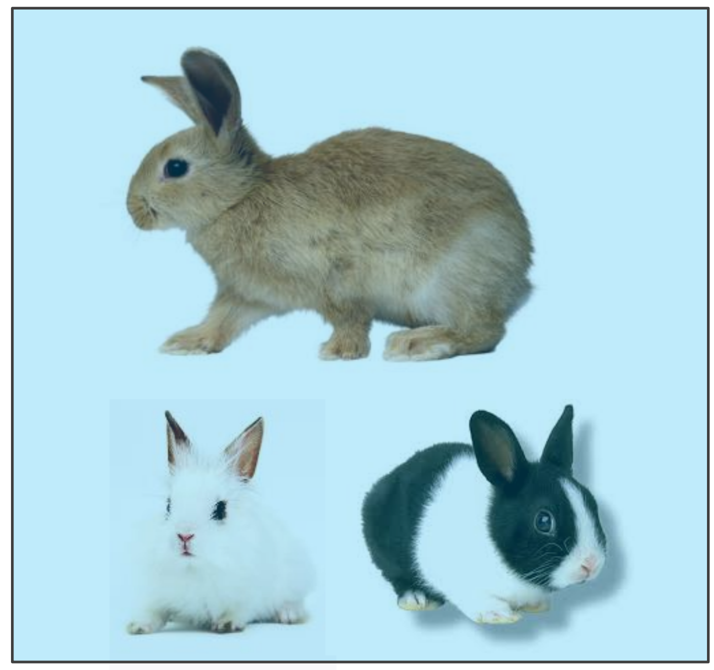
- Instance has the **structure of the class**

WHY USE OOP AND CLASSES OF OBJECTS?

- Model or simulate real life – systems of objects

WHY USE OOP AND CLASSES OF OBJECTS?

- Model or simulate real life – systems of objects
- Group different objects of the same type; capture common patterns of use



GROUPS OF OBJECTS HAVE ATTRIBUTES (RECAP)

- **Data attributes**
  - How can you represent your object with data?
  - **What it is**
  - *for a coordinate: x and y values*
  - *for an animal: age, name*
- **Procedural attributes** (behavior/operations/**methods**)
  - How can someone interact with the object?
  - **What it does**
  - *for a coordinate: find distance between two points*
  - *for an animal: make a sound*

CREATING INSTANCES (Recap)

- Usually when creating an instance of an object type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Method is another name for a procedural attribute, or a procedure that "belongs" to this class

CREATING INSTANCES (Recap)

- Usually when creating an instance of an object type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the instance as the first argument. By convention, we use `self` as the name of the first argument of methods.

CREATING INSTANCES (Recap)

- Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the instance as the first argument. By convention, we use `self` as the name of the first argument of methods.

- The "." operator accesses an attribute of an object, so `__init__` defines two attributes for new object: `x` and `y`.

CREATING INSTANCES (Recap)

- Usually when creating an instance of an object type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When calling a method of an object, Python always passes the instance as the first argument. By convention, we use `self` as the name of the first argument of methods.

- The "." operator accesses an attribute of an object, so `__init__` defines two attributes for new object: `x` and `y`.

When accessing an attribute of an instance, start by looking within the class definition, then move up to the definition of a superclass, eventually move to the global environment

CREATING INSTANCES (Recap)

- Usually when creating an instance of an object type, we will want to provide some initial values for the internal data. To do this, define an `__init__` method:

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x, origin.x)
```

The expression
`classname(values…)`
creates a new object of type classname and then calls its `__init__` method with the new object and `values…` as the arguments. When the method is finished executing, Python returns the initialized object as the value.

Note that don't provide argument for self, Python does this automatically

VISUALIZING THIS IDEA

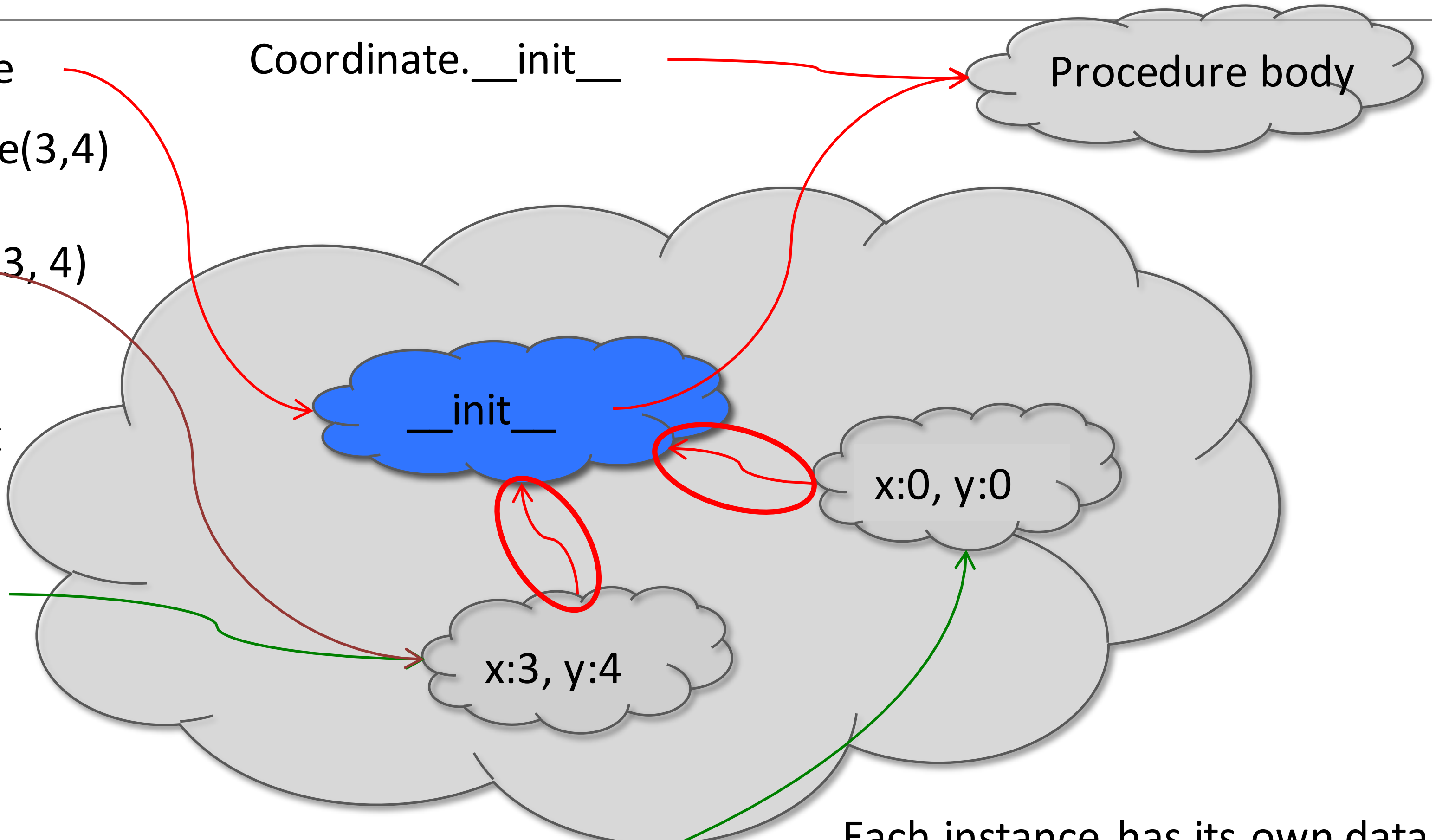
Coordinate

Coordinate.__init__

c = Coordinate(3,4)
Calls
__init__(self, 3, 4)

This calls
self.x = x
self.y = y

So c points to



origin = Coordinate(0, 0)
Then origin points to

Each instance has its own data attributes, both instances inherit attributes of Coordinate class

DEFINING ANIMAL CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None

myanimal = Animal(3)
```

class definition

name

class parent

variable to refer to an instance of the class

what data initializes an Animal type

bind local age variable to input value

special method to create an instance

name is a data attribute even though an instance is not initialized with it as a parameter

one instance

mapped to self.age in class def

GETTER AND SETTER METHODS (RECAP)

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getters

setters

print

- **Getters and setters** should be used outside of class to access data attributes

AN INSTANCE and DOT NOTATION (RECAP)

- Instantiation creates an **instance of an object**

```
a = Animal(3)
```

- **Dot notation** can be used to access attributes (data and methods) though it is better to use getters and setters to access data attributes

```
a.age
```

```
a.get_age()
```

- access data attribute directly
- allowed, but NOT recommended

- access method
- best to use getters and setters

INFORMATION HIDING

- Author of class definition may **change data attribute** variable names

```
class Animal(object):
    def __init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

replaced age data attribute by years

- If you are **accessing data attributes** outside the class and class **definition changes**, may get errors
- Outside of class, use getters and setters instead use `a.get_age()` NOT `a.age`
  - good style
  - easy to maintain code
  - prevents bugs

Best to only access or change attributes of an instance by using methods – isolates details of instance from use of instance

PYTHON NOT GREAT AT INFORMATION HIDING

- Allows you to **access data** from outside class definition in an instance

```
print(a.age)
```

- Allows you to **write to data** from outside class definition to an instance

```
a.age = 'infinite'
```

- Allows you to **create data attributes** for an instance from outside class definition

```
a.size = "tiny"
```

- It's **NOT GOOD STYLE** to do any of these!

DEFAULT ARGUMENTS

- **Default arguments** for formal parameters are used if no actual argument is given

```
def set_name(self, newname=""):
    self.name = newname
```

- Default argument used here

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

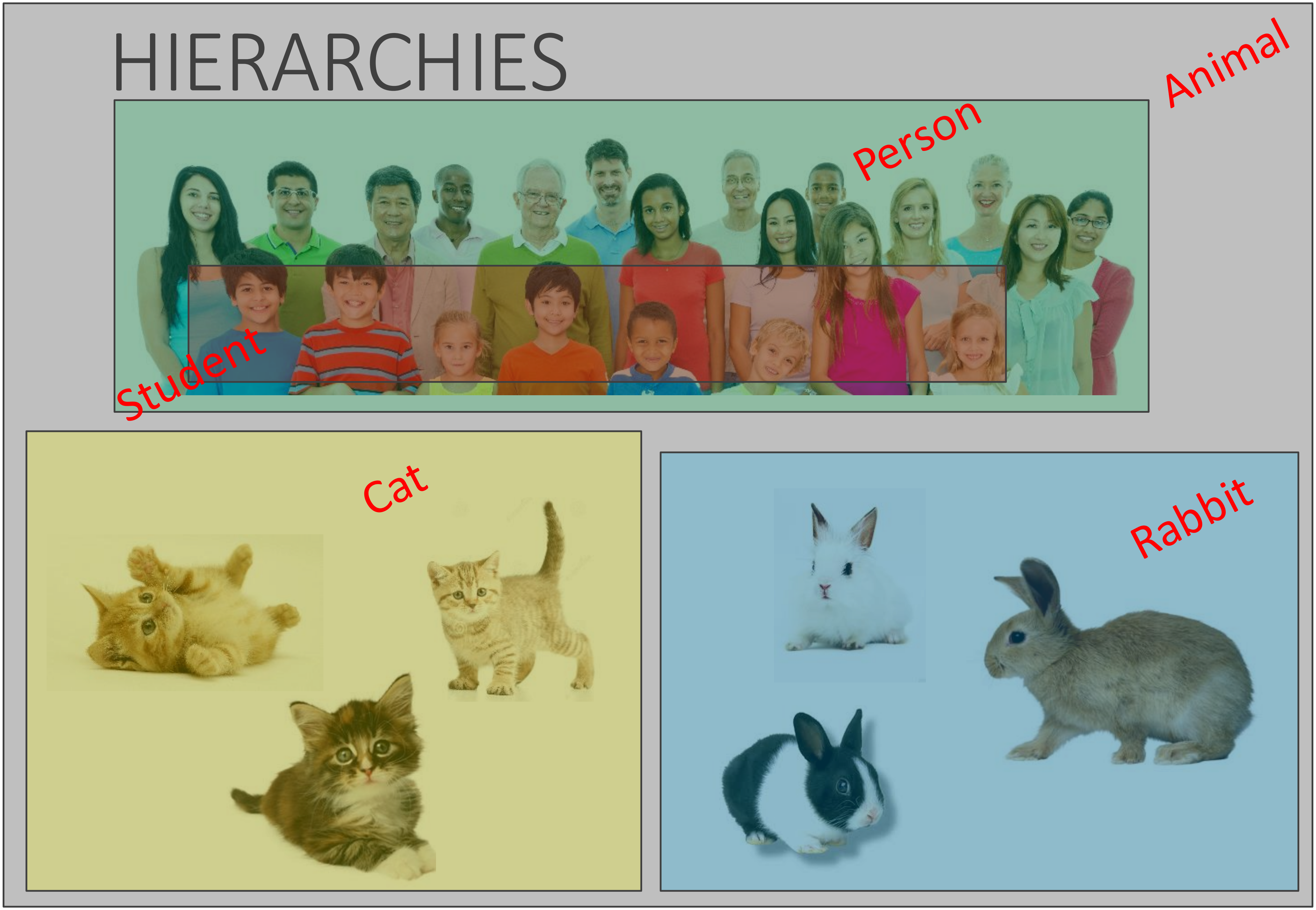
prints ""

- Argument passed in is used here

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

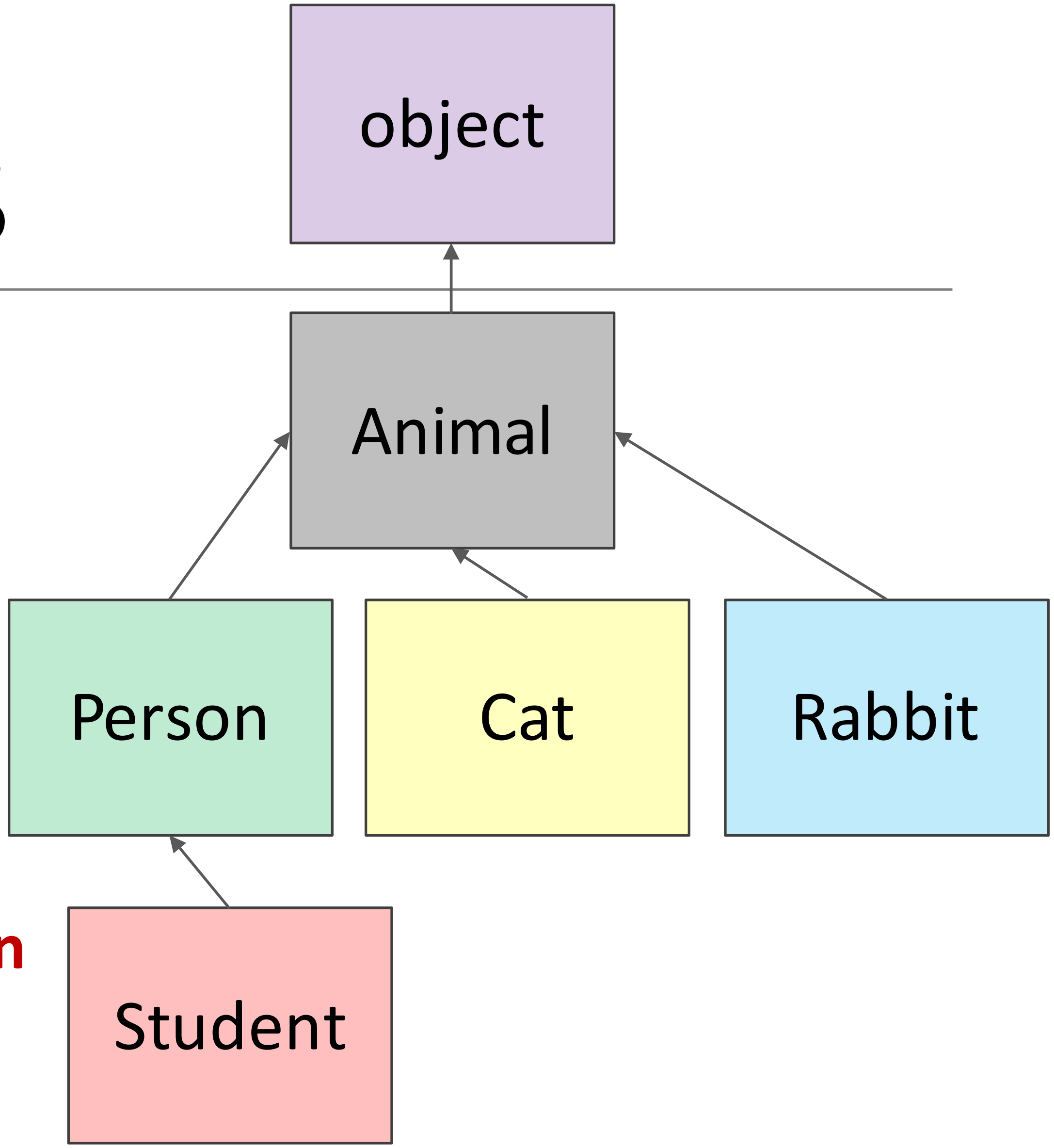
prints "fluffy"

HIERARCHIES



HIERARCHIES

- **Parent class** (superclass)
- **Child class** (subclass)
  - **Inherits** all data and behaviors of parent class
  - **Add** more **information**
  - **Add** more **behaviors**
  - **Override** behavior



INHERITANCE: PARENT CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- everything is an object
- class object implements basic operations in Python, like binding variables, etc.

INHERITANCE: SUBCLASS

inherits all attributes of Animal:
__init__()
age, name
get_age(), get_name()
set_age(), set_name()
__str__()

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)
```

add new functionality via speak method

overrides Animal's __str__ method

- Add new functionality with `speak()`
  - Instance of type `Cat` can be called with new methods
  - Instance of type `Animal` throws error if called with `Cat`'s new method
- `__init__` is not missing, uses the `Animal` version

USING THE HIERARCHY

```
In [31]: jelly = Cat(1)
In [32]: jelly.set_name('JellyBelly')
In [33]: print(jelly)
cat:JellyBelly:1

In [34]: print(Animal.__str__(jelly))
animal:JellyBelly:1

In [35]: blob = Animal(1)
In [36]: print(blob)
animal:None:1

In [37]: blob.set_name()
In [38]: print(blob)
animal::1
```

inherits method from `Animal`

Cat `__str__` method shadows method from `Animal`

could explicitly recover underlying `Animal` method

can change values of attributes of an instance

Animal is a class	. gets associated attribute	In this case, `__str__` method for Animal

INHERITANCE

```
class Cat(Animal):
    def speak(self):
        print("meow”)
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)

class Rabbit(Animal):
    def speak(self):
        print(”meep”)
    def __str__(self):
        return ”rabbit:"+str(self.name)+":"+str(self.age)
```

Different subclasses of Animal can specialize methods and attributes (like speak) while inheriting common methods and attributes (like age and name)

USING THE HIERARCHY

```
In [31]: jelly = Cat(1)
In [34]: blob = Animal(1)
In [38]: peter = Rabbit(5)
In [39]: jelly.speak()
meow

In [40]: peter.speak()
meep

In [41]: blob.speak()
AttributeError: 'Animal' object has no
attribute 'speak'
```

uses method from Cat

uses method from Rabbit

tries to find method in Animal

WHICH METHOD TO USE?

- Subclass can have **methods with same name** as superclass or other subclass
- For an instance of a class, look for a method of that name in **current class definition**
- If not found, look for method of that name **up the hierarchy** (in parent, then grandparent, and so on)
- Use first method in the hierarchy that you found with that method name

```python
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends.copy()
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

parent class is Animal

call Animal constructor

call Animal's method

add a new data attribute

use copy to avoid mutation

new methods

override Animal's __str__ method

Calling Animal constructor gets all attributes of superclass; rest of this __init__ method just adds attributes specific to subclass or instance

USING THE HIERARCHY


```
In [42]: alice = Person('Alice', 29)
In [43]: tarrant = Person('Tarrant', 56)
In [44]: alice.speak()
hello

In [45]: alice.age_diff(tarrant)
27 year difference

In [46]: Person.age_diff(tarrant,alice)
27 year difference
```

uses method from Person

uses method associated with instance

can use class attribute to find method

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

bring in functions from random library

inherits Person and Animal attributes

uses Person __init__ which uses Animal __init__

adds new data

random() method gives back float in [0, 1)

USING THE HIERARCHY


```
In [42]: alice = Person('Alice', 45)
In [47]: tarrant = Student('Tarrant', 18, 'Course VI')
In [48]: print(tarrant)
student:Tarrant:18:Course VI

In [49]: tarrant.speak()
i have homework
In [50]: tarrant.speak()
i have homework
In [51]: tarrant.speak()
i am watching tv
In [52]: tarrant.speak()
i should eat
```

uses method from Student

uses method from Student

if called multiple times, may get different behavior because of random

INSTANCE VARIABLES vs CLASS VARIABLES

- we have seen **instance variables** so far in code
- specific to an instance
- created for **each instance**, belongs to an instance
- used the generic variable name self within the class definition

`self.variable_name`

- introduce **class variables** that belong to the class
- defined inside class but outside any class methods, outside `__init__`
- **shared** among all objects/instances of that class

RECALL `Animal` CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

CLASS VARIABLES AND THE `Rabbit` SUBCLASS

- **class variables** and their values are shared between all instances of a class

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

parent class

class variable

instance variable

access class variable

incrementing class variable changes it for all instances that may reference it

- `tag` used to give **unique id** to each new rabbit instance

Rabbit GETTER METHODS

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(5)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
    def __str__(self):
        return "rabbit:"+ self.get_rid()
```

method on a string to pad the beginning with zeros for example, 00001 not 1

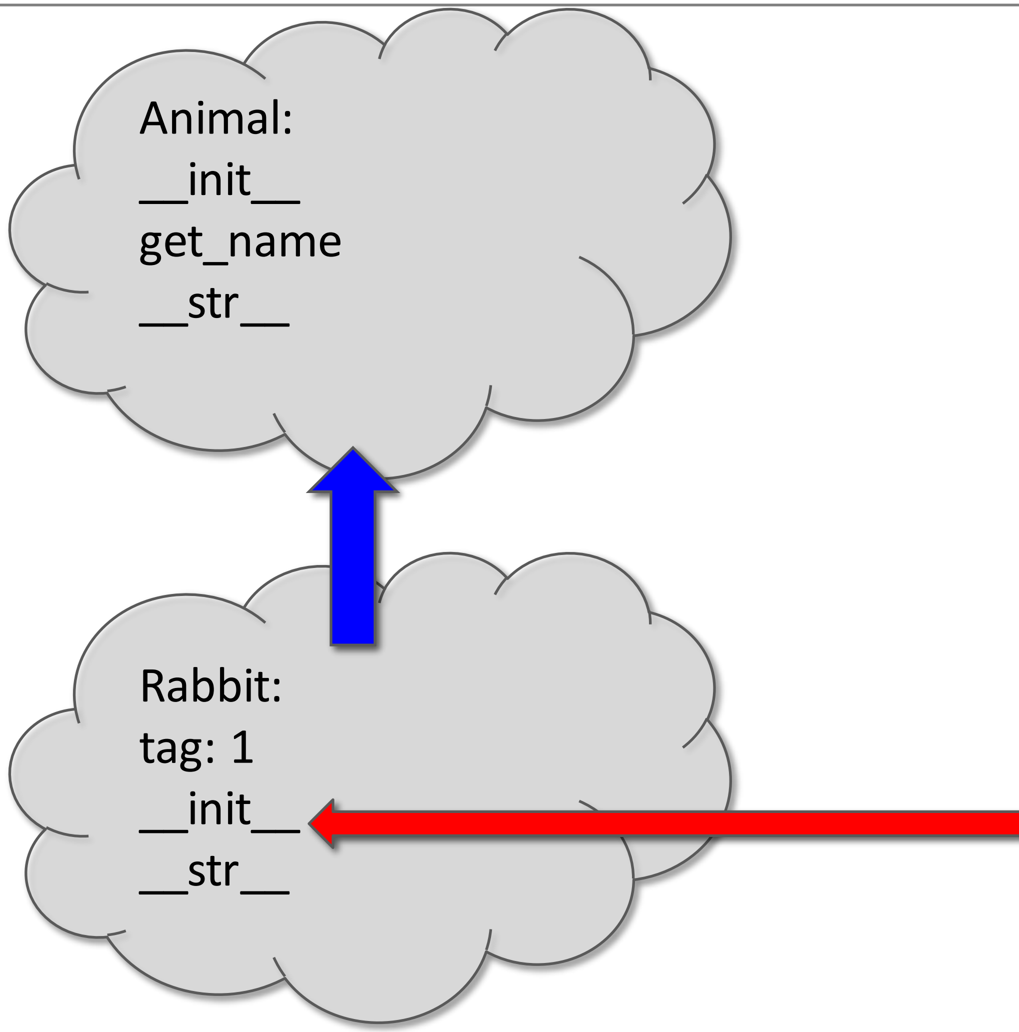
- getter methods specific for a Rabbit class
- there are also getters get_name and get_age inherited from Animal

Return actual object, in this case a Rabbit

EXAMPLE USAGE

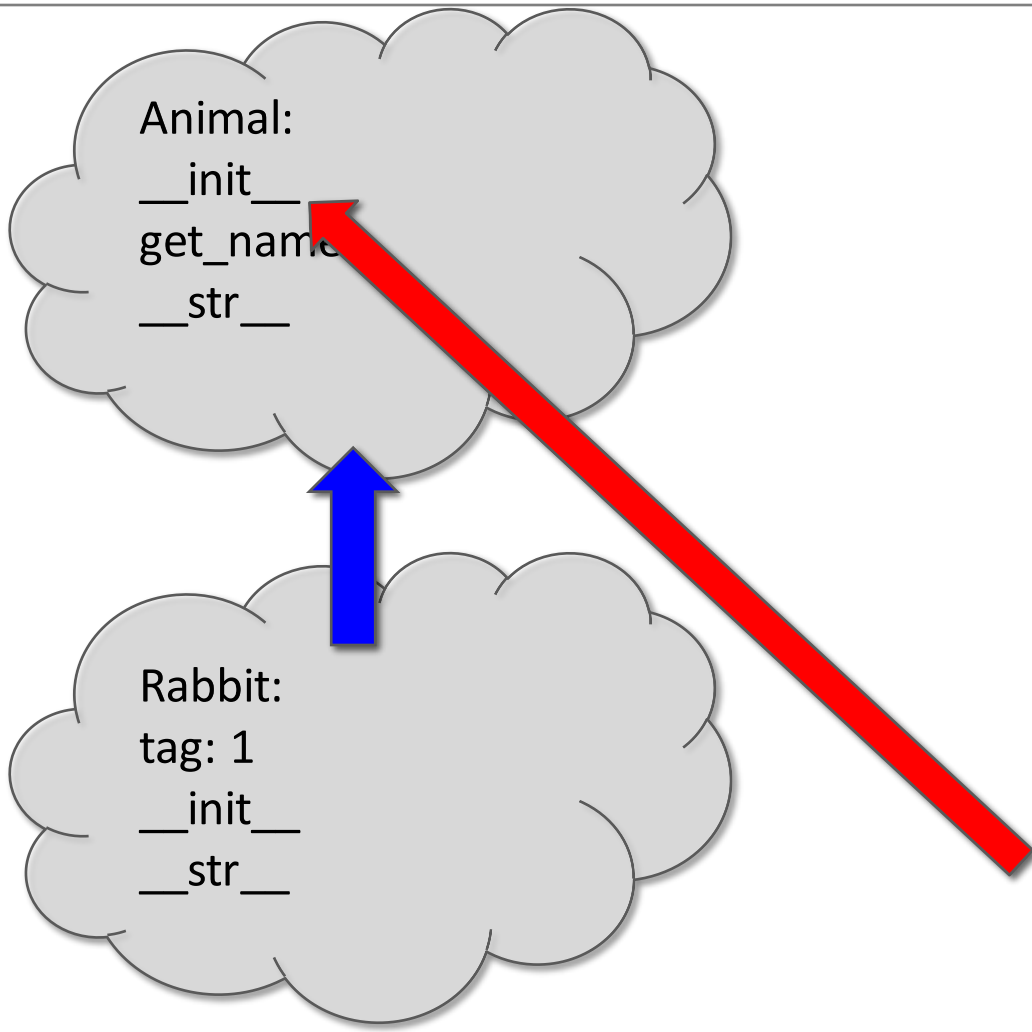

```
In 35: r1 = Rabbit(3)
In 36: r2 = Rabbit(1)
In 37: r3 = Rabbit(10)

In 38: print("r1:", r1)
r1: rabbit:00001

In 39: print("r2:", r2)
r2: rabbit:00002

In 40: print("r1 parent1:", r1.get_parent1())
r1 parent1: None
```

VISUALIZING THE HIERARCHY



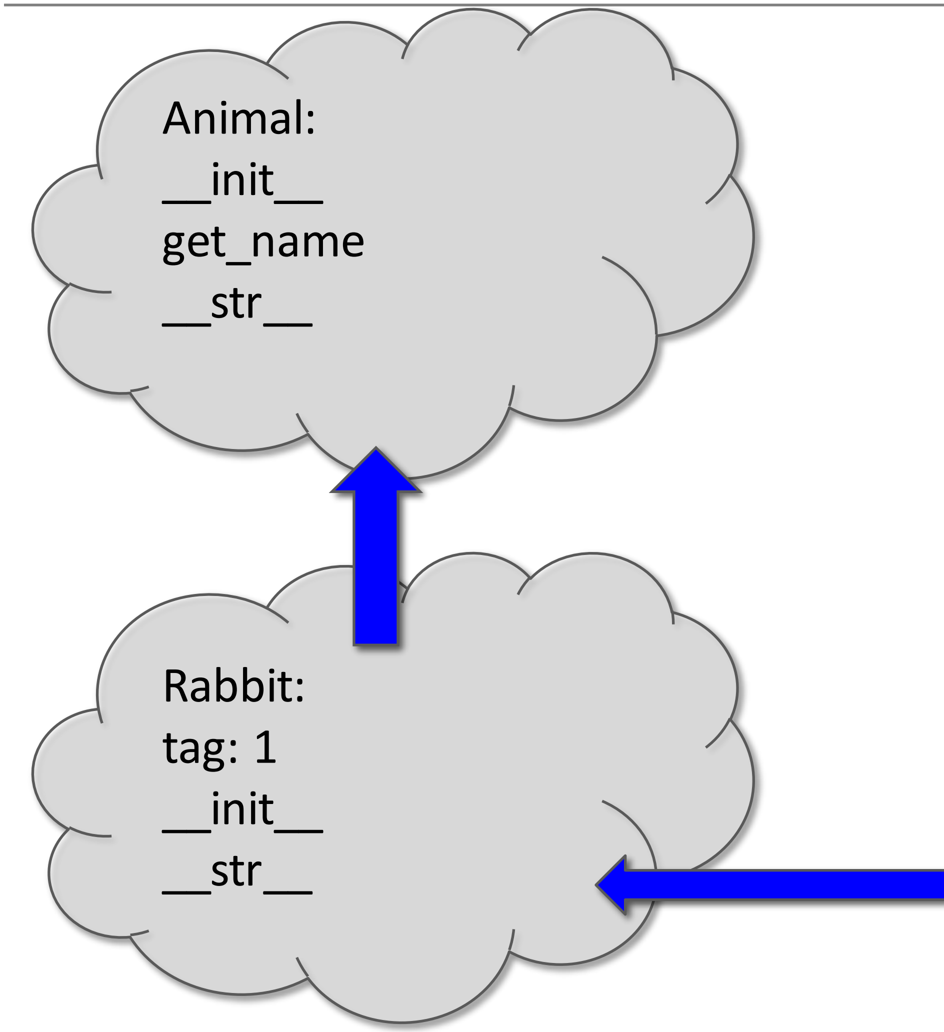
Calling Rabbit will create an instance and use this __init__ procedure on it (because that is the one visible in Rabbit's environment)

VISUALIZING THE HIERARCHY



That code uses
Animal.__init__
which will in turn call this
procedure on the
instance

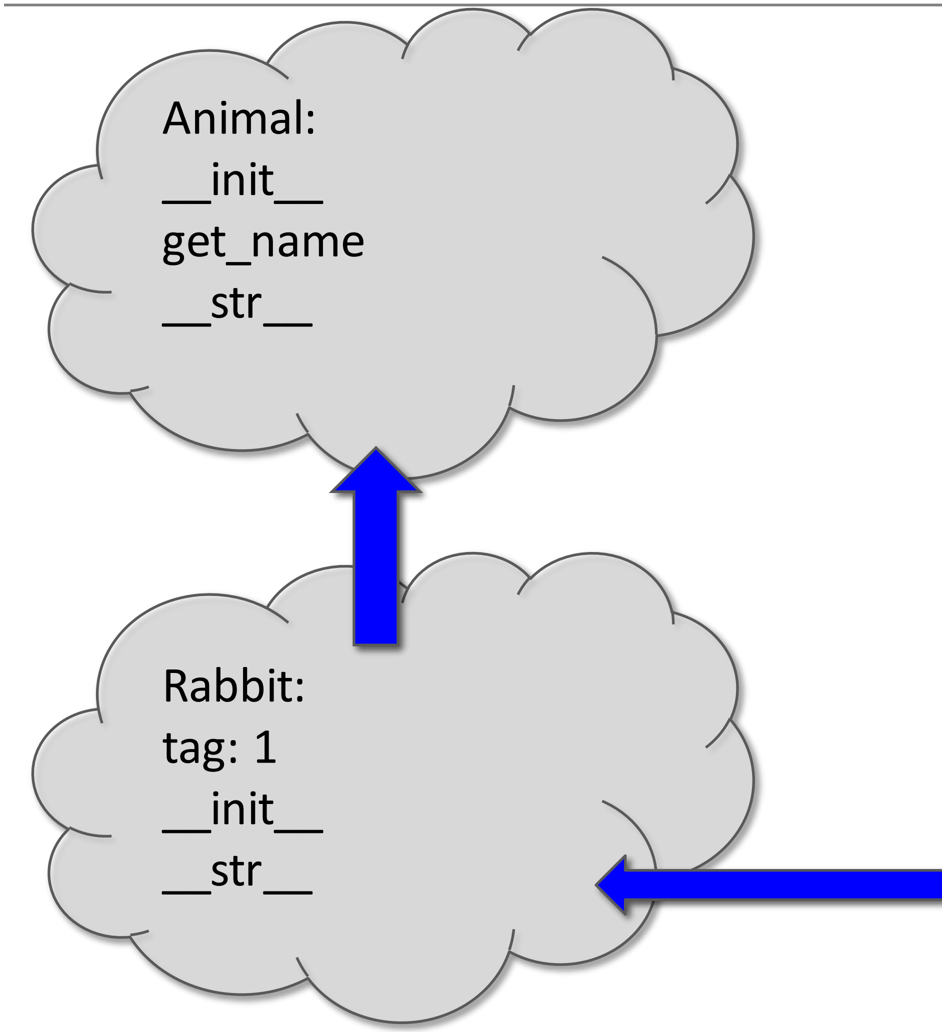
VISUALIZING THE HIERARCHY



And thus the instance of Rabbit (because of the first call, which inherits from the class definition) will have bindings set by the inherited __init__ code

name	
age	3
name	None

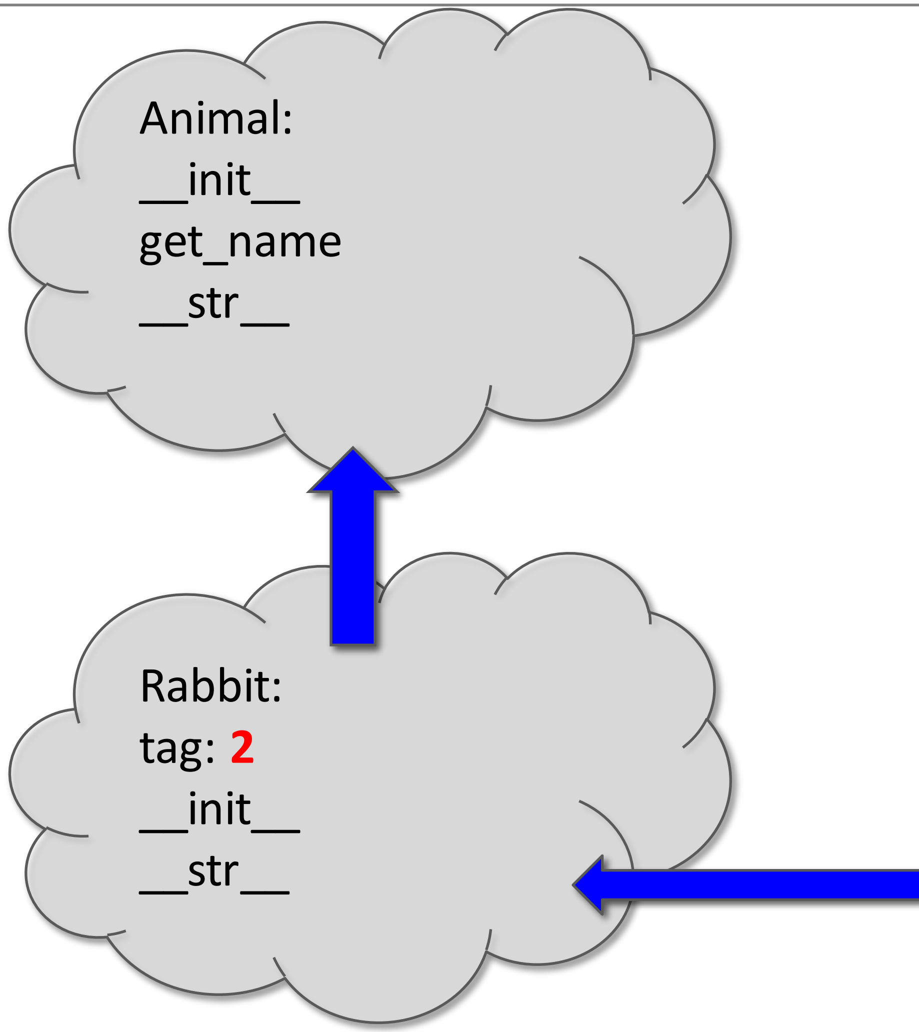
VISUALIZING THE HIERARCHY



The rest of the original __init__ code calls
self.rid = Rabbit.tag
which creates a binding in self (i.e. the instance) to current value of tag (in class)

name	
age	3
name	None
rid	1

VISUALIZING THE HIERARCHY

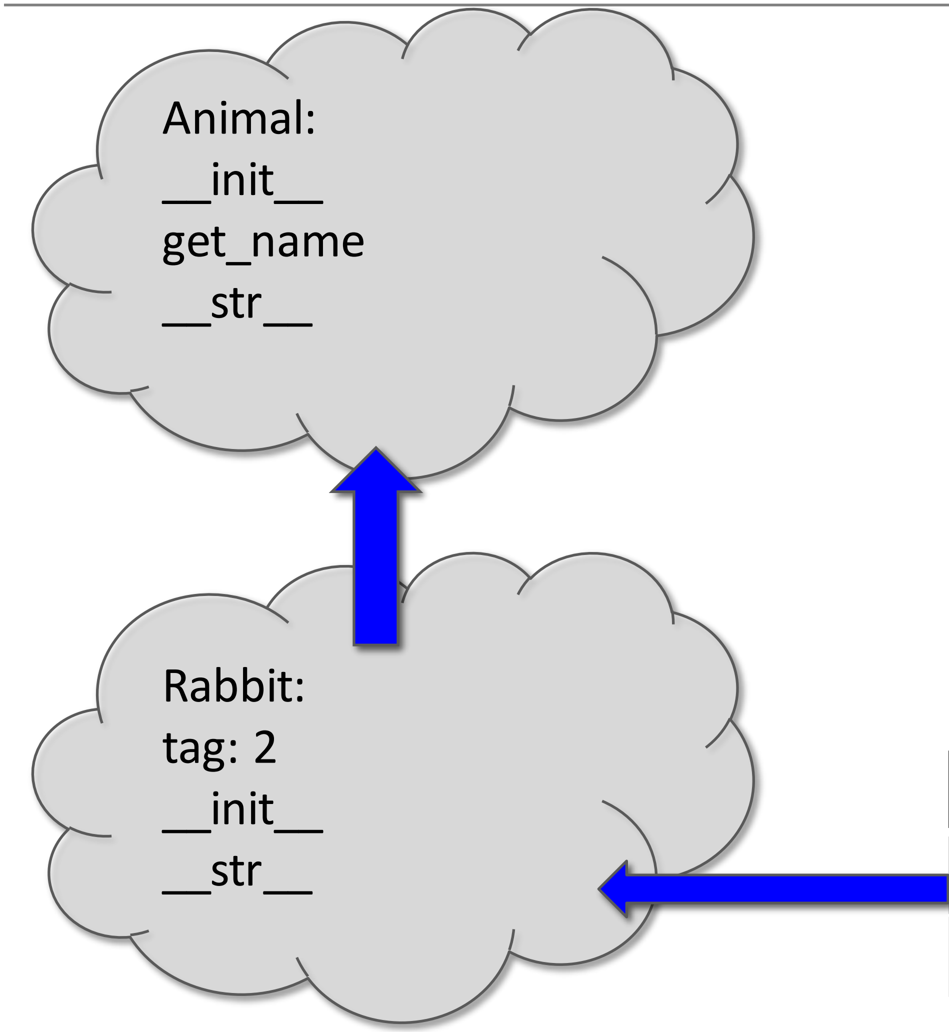


The rest of the original __init__ code calls
self.rid = Rabbit.tag
which creates a binding in self (i.e. the instance) to current value of tag (in class)

And then calls
Rabbit.tag += 1

name	
age	3
name	None
rid	**1**

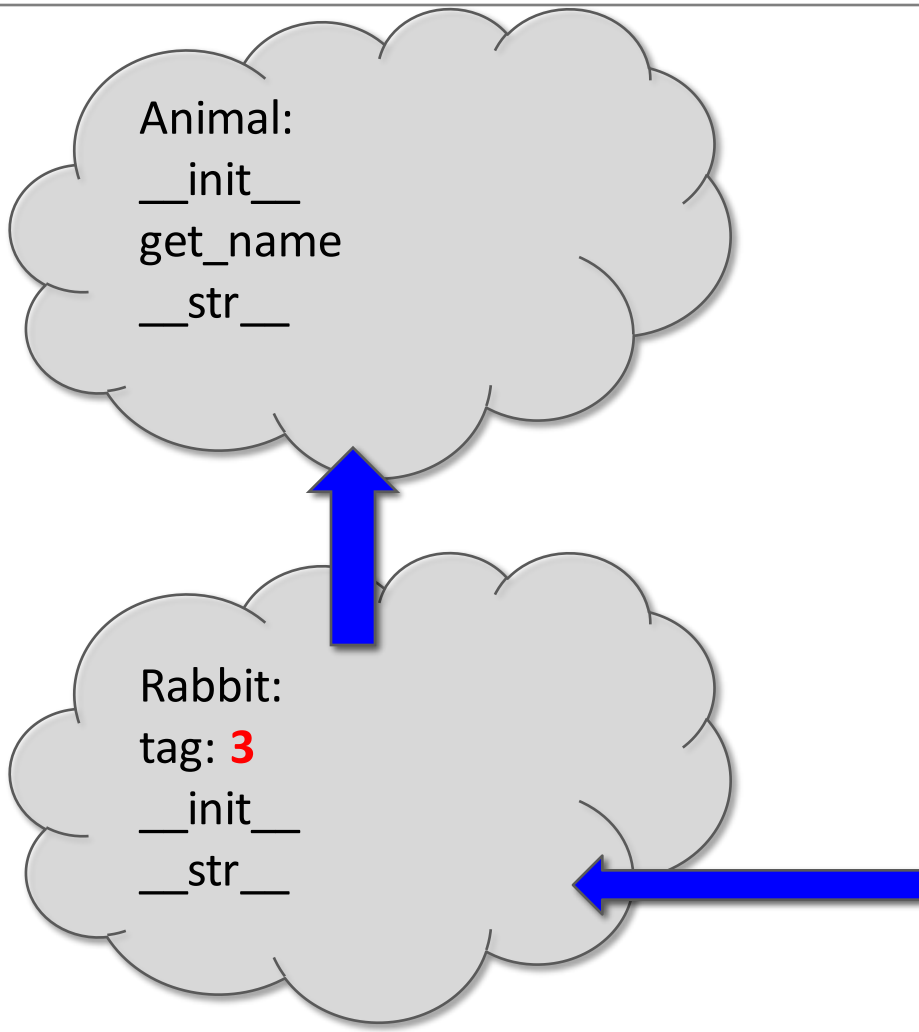
VISUALIZING THE HIERARCHY



Thus calling Rabbit a second time to create a second instance will execute the same sequence, but now tag is bound to 2

name	
age	1
name	None

VISUALIZING THE HIERARCHY



And this will create a new instance with a unique id number

name	
age	1
name	None
rid	**2**

WORKING WITH YOUR OWN TYPES

```
def __add__(self, other):
    # returning object of same type as this class
    return Rabbit(0, self, other)
```

recall Rabbit's `__init__(self, age, parent1=None, parent2=None)`

- Define **+ operator** between two `Rabbit` instances
  - Define what something like this does: `r4 = r1 + r2` where `r1` and `r2` are `Rabbit` instances
  - `r4` is a new `Rabbit` instance with age 0
  - `r4` has `self` as one parent and `other` as the other parent
  - In `__init__`, **`parent1` and `parent2` are of type `Rabbit`**

EXAMPLE USAGE

```
In [53]: peter = Rabbit(2)
In [54]: peter.set_name('Peter')
In [55]: hopsy = Rabbit(3)
In [56]: hopsy.set_name('Hopsy')
In [61]: mopsy = peter + hopsy
In [62]: mopsy.set_name('Mopsy')
In [63]: print(mopsy.get_parent1())
rabbit:00007

In [64]: mopsy.get_parent1().get_name()
Out[64]: 'Peter'
```

Need these to get actual object	Then access instance's data

SPECIAL METHOD TO COMPARE TWO `Rabbits`

- Decide that two rabbits are equal if they have the **same two parents**

```
def __eq__(self, other):
    parents_same = self.parent1.rid == other.parent1.rid \
                   and self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \
                       and self.parent1.rid == other.parent2.rid
    return parents_same or parents_opposite
```

booleans

- Compare ids of parents since **ids are unique** (due to class var)
- Note you can't compare objects directly
  - For example, can't try `self.parent1 == other.parent1`
  - This calls the `__eq__` method over and over until call it on `None` and gives an `AttributeError` when it tries to do `None.parent1`

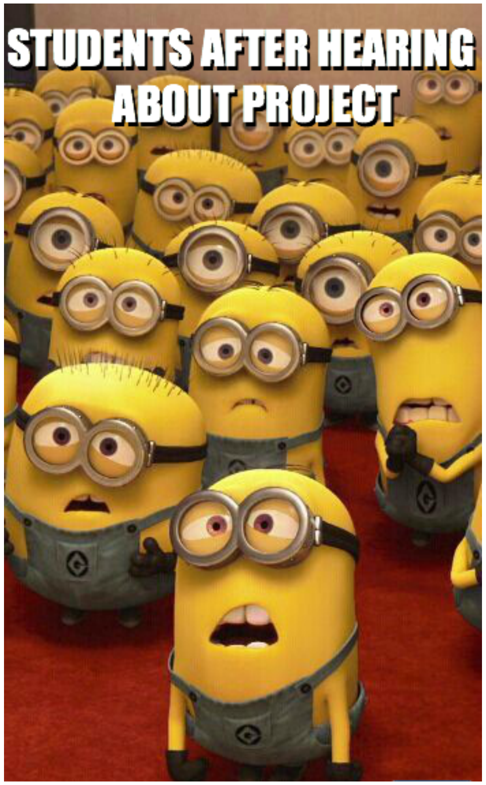
EXAMPLE USAGE

```
In [53]: peter = Rabbit(2)
In [54]: peter.set_name('Peter')
In [55]: hopsy = Rabbit(3)
In [56]: hopsy.set_name('Hopsy')
In [57]: cotton = Rabbit(1, peter, hopsy)
In [58]: cotton.set_name('Cottontail')
In [61]: mopsy = peter + hopsy
In [62]: mopsy.set_name('Mopsy')

In [65]: print(mopsy == cotton)
True
```

THE POWER OF OBJECT ORIENTED PROGRAMMING

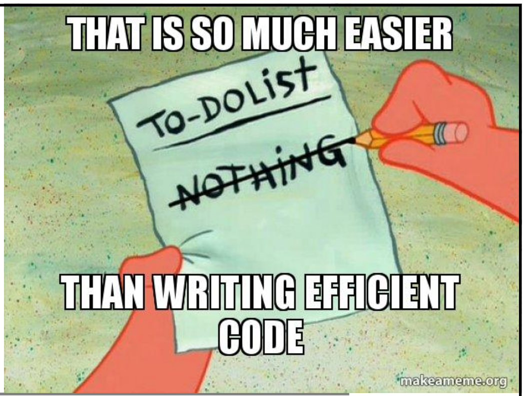
- **Bundle together objects** that share
  - common attributes and
  - procedures that operate on those attributes
- Use **abstraction** to make a distinction between how to implement an object versus how to use an object
- Build **layers** of object abstractions that inherit behaviors from other classes of objects
- Create our **own classes of objects** on top of Python's basic classes

5 Minute Break

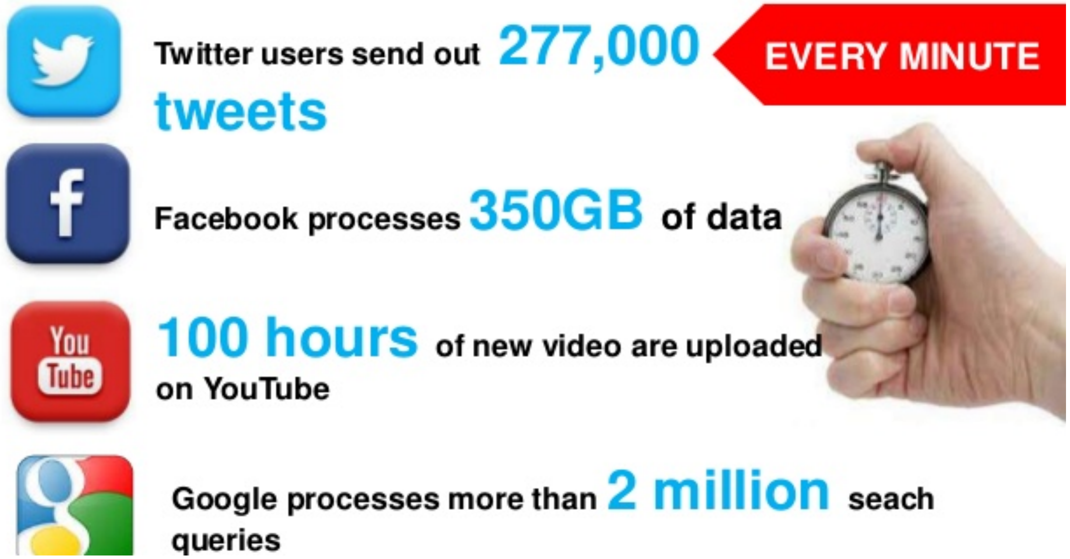


PROGRAM EFFICIENCY

WRITING EFFICIENT PROGRAMS



- So far, we have emphasized correctness. It is the first thing to worry about!
- But sometimes that is not enough
- Problems can be very complex (as we shall see when we get to optimization in 6.0002)
- But data sets can be very large: in 2014 Google served 30,000,000,000,000 pages covering 100,000,000 GB of data



EFFICIENCY IS IMPORTANT

- Separate **time and space efficiency** of a program
- Tradeoff between them: can use up a bit more memory to store values for quicker lookup later
- Challenges in understanding efficiency
  - A program can be **implemented in many different ways**
  - You can solve a problem using only a handful of different **algorithms**
- Want to separate choice of implementation from choice of more abstract algorithm

EVALUATING PROGRAMS

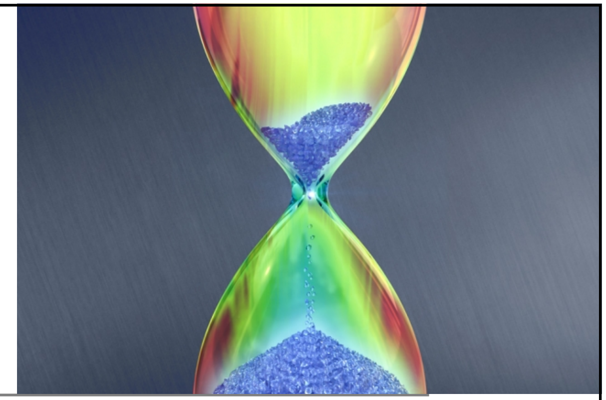
- Measure with a **timer**
- **Count** the operations
- Abstract notion of **order of growth**

TIMING A PROGRAM

- Use time module
- Recall that importing means to bring in that class into your own file
- **Start** clock →
- **Call** function →
- **Stop** clock →

```
import time

def c_to_f(c):
    return c*9.0/5 + 32

t0 = time.clock()
c_to_f(100000)
t1 = time.clock() - t0
print("t =", t1, "s,")
```

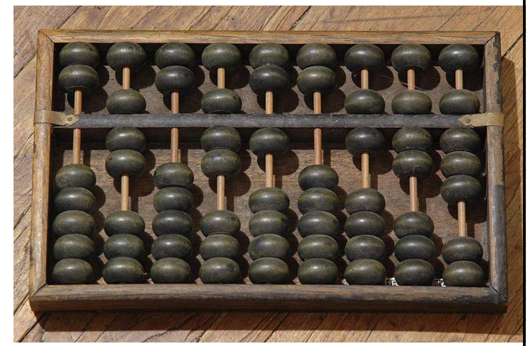
TIMING PROGRAMS IS INCONSISTENT



- GOAL: to evaluate different algorithms
- Running time **varies between algorithms** ✓
- Running time **varies between implementations** ✗
- Running time **varies between computers** ✗
- Running time is **not predictable** for small inputs ✗

- Time varies for different inputs but cannot really express a relationship between inputs and time ✗

COUNTING OPERATIONS



- Assume these steps take **constant time**:
  - Mathematical operations
  - Comparisons
  - Assignments
  - Accessing objects in memory
- Count number of operations executed as function of size of input

```
def c_to_f(c):
    return c*9.0/5 + 32
```

3 ops

```
def mysum(x):
    total = 0
    for i in range(x+1):
        total += i
    return total
```

1 op

loop x times

1 op

2 ops

mysum → 1+3(x+1) ops

COUNTING OPERATIONS IS BETTER, BUT …

- GOAL: to evaluate different algorithms
- Count **depends on algorithm** ✓
- Count **depends on implementations** ✗
- Count **independent of computers** ✓
- No real definition of **which operations** to count ✗

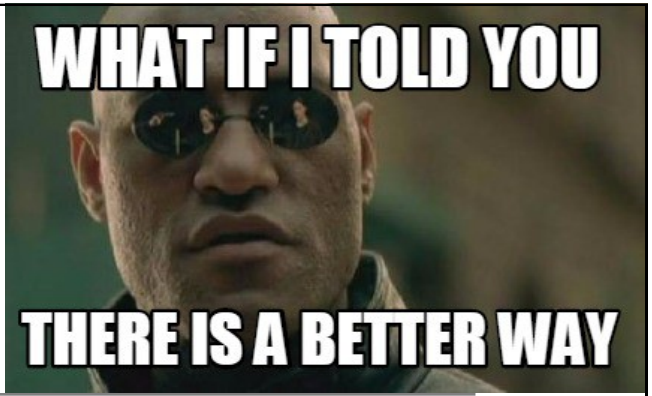
- Count varies for different inputs and can come up with a relationship between inputs and the count ✓

… STILL NEED A BETTER WAY

- Timing and counting **evaluate implementations**
- Timing and counting **evaluate machines**

- Want to **evaluate algorithm**
- Want to **evaluate scalability**
- Want to **evaluate in terms of input size**

A BETTER WAY



- Focus on idea of counting operations in an algorithm, but **not worry about small variations in implementation**
- Focus on how algorithm performs when **size of problem gets arbitrarily large**
- Want to **relate time** needed to complete a computation, measured this way, **against the size of the input** to the problem
- Need to decide what to measure, given that actual number of steps may depend on specifics of trial

HOW TO CHOOSE WHICH INPUT TO USE TO EVALUATE A FUNCTION

- Want to express **efficiency in terms of input**, so need to decide what is your input
- Could be an **integer**
  -- `mysum(x)`
- Could be **length of list**
  -- `list_sum(L)`
- **You decide** when multiple parameters to a function
  -- `search_for_elmt(L, e)`

DIFFERENT INPUTS CHANGE HOW THE PROGRAM RUNS

- A function that searches for an element in a list

```
def search_for_elmt(L, e):
    for i in L:
        if i == e:
            return True
    return False
```

- When `e` is **first element** in the list → BEST CASE
- When `e` is **not in list** → WORST CASE
- When **look through about half** of the elements in list → AVERAGE CASE
- Want to measure this behavior in a general way

BEST, AVERAGE, WORST CASES

- Consider that you are given a list `L` of some length `len(L)`
- **Best case**: minimum running time over all possible inputs of a given size, `len(L)`
  - Constant for `search_for_elmt`
  - First element in any list
- **Average case**: average running time over all possible inputs of a given size, `len(L)`
  - Practical measure
- **Worst case**: maximum running time over all possible inputs of a given size, `len(L)`
  - Linear in length of list for `search_for_elmt`
  - Must search entire list and not find it
  - Focus on **worst case** in this class

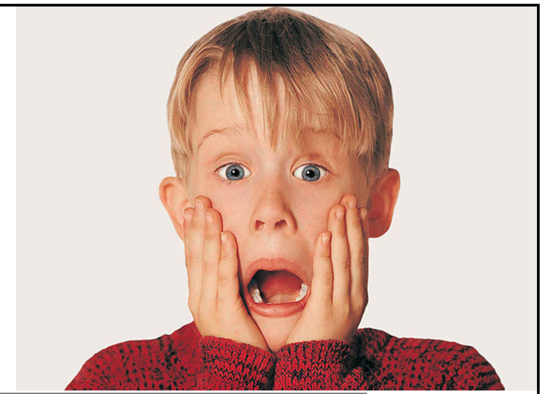
ORDERS OF GROWTH

- Want to evaluate programs when **input is very big**
- Want to express the **growth of program's run time**
- Want to put an **upper bound** on growth
- Do not need to be precise: **"order of" not "exact"** growth
- We will look at **largest factors** in run time (which section of the program will take the longest to run?)

MEASURING ORDER OF GROWTH: BIG O() NOTATION

- Big Oh notation measures an **upper bound on the asymptotic growth**, often called order of growth
- **Big Oh or O()** is used to describe worst case
  - Worst case occurs often and is the bottleneck when a program runs
  - Express rate of growth of program relative to the input
  - Evaluate algorithm not machine or implementation
- A technicality
  - When we say that the complexity of f is O(n), we mean that its asymptotic growth is not worse than linear in n.
  - It is an **upper bound**, not necessarily a **tight bound**
  - In practice, we are usually looking for something close to a tight bound

EXACT STEPS vs O()



```
def fact_iter(n):
    """assumes n an int >= 0"""
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer
```

temp = n-1
n = temp

- Computes factorial
- Number of steps: 1 + 7n + 1
- Worst case asymptotic complexity: O(n)
  - Ignore additive constants
  - Ignore multiplicative constants

WHAT DOES O(N) MEASURE?

- Interested in describing how amount of time needed grows as size of (input to) problem grows
- Given an expression for the number of operations needed to compute an algorithm, want to know **asymptotic behavior as size of problem gets large**
- Will focus on term that grows most rapidly
- Ignore multiplicative constants, since want to know how rapidly time required increases as increase size of input

SIMPLIFICATION EXAMPLES

- Drop constants and multiplicative factors
- Focus on **dominant term**

$O(n^2)$: $n^2 + 2n + 2$

$O(n^2)$: $n^2 + 100000n + 3^{1000}$

$O(n)$: $\log(n) + n + 4$

ANALYZING PROGRAMS AND THEIR COMPLEXITY

- **Combine** complexity classes
  - Analyze statements inside functions
  - Apply some rules, focus on dominant term

Law of Addition for O():

- Used with **sequential** statements
- O(f(n)) + O(g(n)) is O(f(n) + g(n))
- For example,

```
for i in range(n):
    print('a')
for j in range(n*n):
    print('b')
```

O(n)

$O(n^2)$

is $O(n) + O(n*n) = O(n+n^2) = O(n^2)$ because of dominant term

ANALYZING PROGRAMS AND THEIR COMPLEXITY

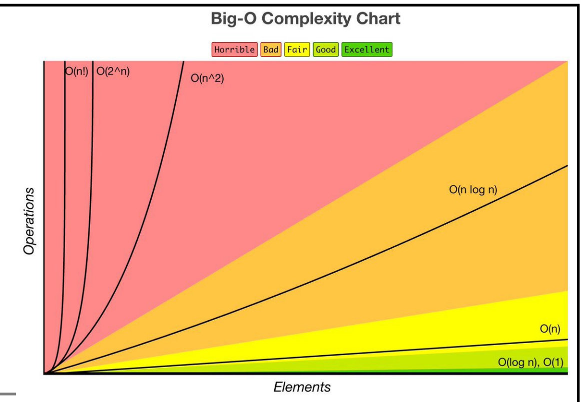
- **Combine** complexity classes
  - Analyze statements inside functions
  - Apply some rules, focus on dominant term

Law of Multiplication for O():

- Used with **nested** statements/loops
- O(f(n)) * O(g(n)) is O(f(n) * g(n))
- For example,

```
for i in range(n):          O(n)
    for j in range(n):      O(n) for each outer loop iteration
        print 'a'
```

is $O(n)*O(n) = O(n*n) = O(n^2)$ because the outer loop goes n times and the inner loop goes n times for every outer loop iter.

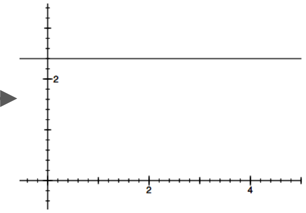


COMPLEXITY CLASSES

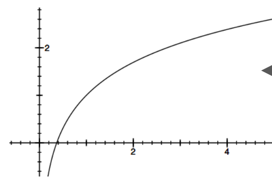
- *O(1)* denotes **constant** running time
- *O(log n)* denotes **logarithmic** running time
- *O(n)* denotes **linear** running time
- *O(n log n)* denotes **log-linear** running time
- $O(n^c)$ denotes **polynomial** running time (c is a constant)
- $O(c^n)$ denotes **exponential** running time (c is a constant being raised to a power based on size of input)

COMPLEXITY CLASSES ORDERED LOW TO HIGH

`O(1)` : constant →

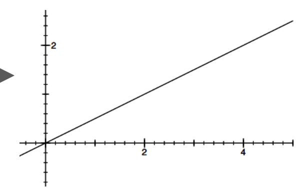


`O(log n)` :

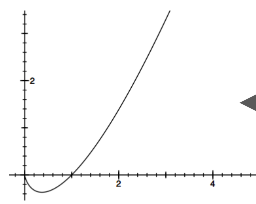


← logarithmic

`O(n)` : linear →

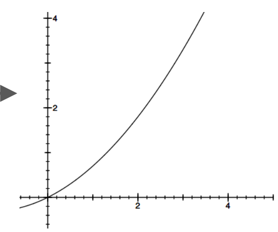


`O(n log n)`:

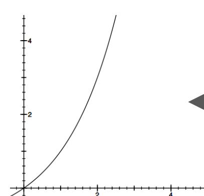


← log linear

`O(n^c)` : polynomial →



`O(c^n)` :



← exponential

c is a constant

COMPLEXITY GROWTH

CLASS	N = 10	N = 100	N = 1000	N = 1000000
O(1)	1	1	1	1
O(log n)	1	2	3	6
O(n)	10	100	1000	1000000
O(n log n)	10	200	3000	6000000
O(n^2)	100	10000	1000000	1000000000000
O(2^n)	1024	12676506 00228229 40149670 3205376	1071508607186267320948425 0490600018105614048117055 3360744375038837035105112 4936122493198378815695858 1275946729175531468251871 4528569231404359845775746 9857480393456777482423098 5421074605062371141877954 1821530464749835819412673 9876755916554394607706291 4571196477686542167660429 8316526243868372056680693 76	Good Luck!!

NEXT TIME

- You will see examples of each of these complexity classes
- You will learn how to recognize algorithmic patterns that are characteristic of each class