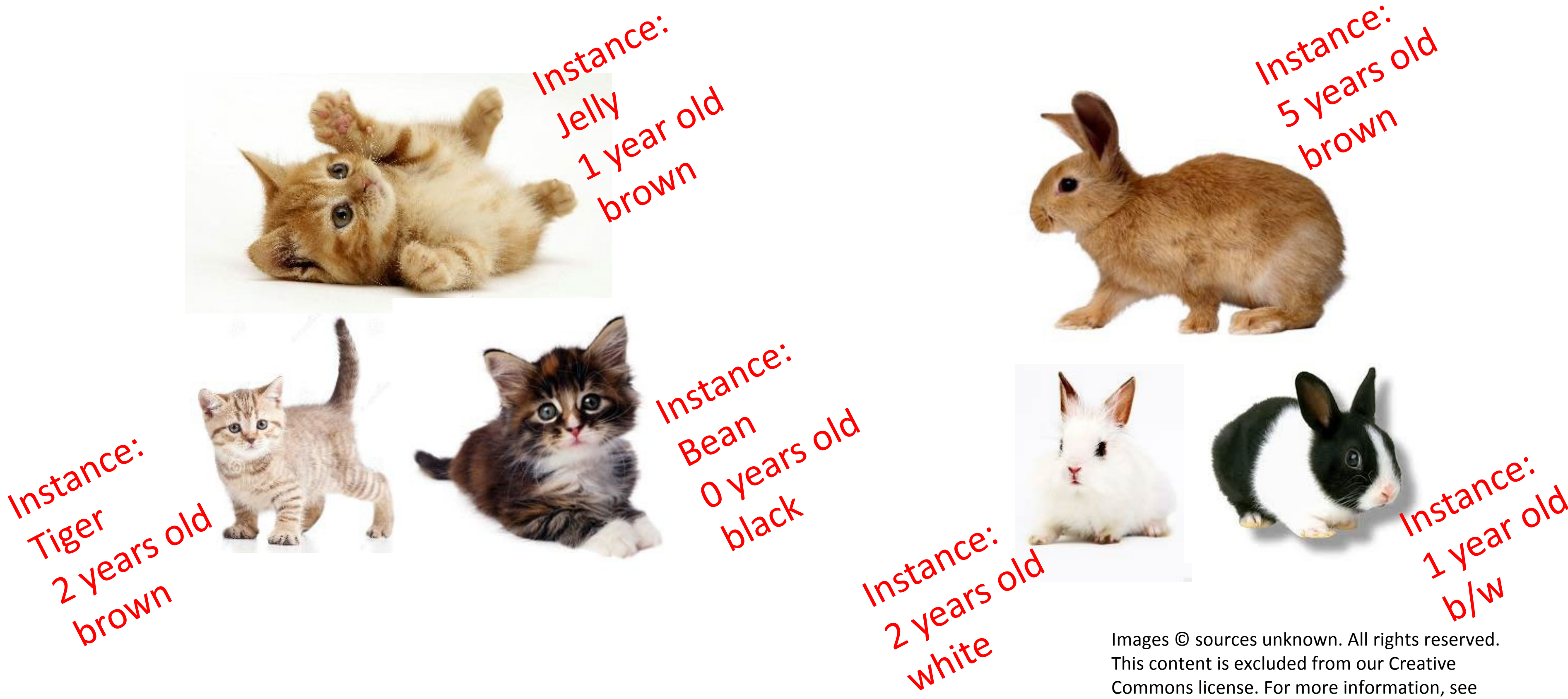
INHERITANCE

WHY USE OOP AND CLASSES OF OBJECTS?

- Mimic real life
- Group different objects part of the same type



WHY USE OOP AND CLASSES OF OBJECTS?

- Mimic real life
- Group different objects part of the same type

All instances of a type have the same data abstraction and behaviors

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
- **Procedural attributes** (behavior/operations/**methods**)
  - How can someone interact with the object?
  - **What it does**

    for a coordinate: find distance between two

    for an animal: print how long it's been alive

HOW TO DEFINE A CLASS (RECAP)

```
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

Special method to create an instance

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

One instance

Mapped to `self.age` in class def

GETTER AND SETTER METHODS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

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

GETTER AND SETTER METHODS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getter

```
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
```

setter

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

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

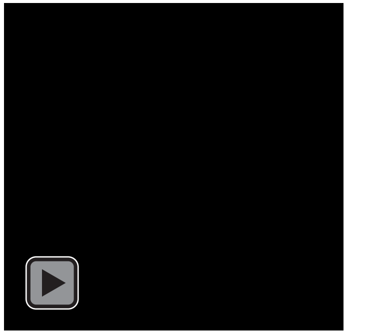
```
a.age
```

- access data attribute
- allowed, but not recommended

```
a.get_age()
```

- 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

CHANGING INTERNAL REPRESENTATION

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

Change internal rep from self.age = age

```
a.get_age()    # works
a.age          # error
```

Accessing methods works correctly, but accessing data attributes no longer works.

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

PYTHON NOT GREAT AT INFORMATION HIDING

- Allows you to **access data** from outside class definition

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

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

```
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!

USE OUR NEW CLASS

```
def animal_dict(L):
    """ L is a list
    Returns a dict, d, mappping an int to an Animal object.
    A key in d is all non-negative ints, n, in L. A value
    corresponding to a key is an Animal object with n as its age. """
    d = {}
    for n in L:
        if type(n) == int and n >= 0:
            d[n] = Animal(n)
    return d

L = [2,5,'a',-5,0]
```

Invoke the name of the class with parameter n (i.e. the age of that animal)

USE OUR NEW CLASS

- Python doesn't know how to call print recursively

```
def animal_dict(L):
    """ L is a list
    Returns a dict, d, mappping an int to an Animal object.
    A key in d is all non-negative ints n L. A value corresponding
    to a key is an Animal object with n as its age. """
    d = {}
    for n in L:
        if type(n) == int and n >= 0:
            d[n] = Animal(n)
    return d

L = [2,5,'a',-5,0]

animals = animal_dict(L)
print(animals)
```

Return is a dict mapping int:Animal

```
{2: <__main__.Animal object at 0x00000199AFF350A0>,
 5: <__main__.Animal object at 0x00000199AFF35A30>,
 0: <__main__.Animal object at 0x00000199AFF35D00>}
```

USE OUR NEW CLASS

```
def animal_dict(L):
    """ L is a list
    Returns a dict, d, mappping an int to an Animal object.
    A key in d is all non-negative ints n L. A value corresponding
    to a key is an Animal object with n as its age. """
    d = {}
    for n in L:
        if type(n) == int and n >= 0:
            d[n] = Animal(n)
    return d

L = [2,5,'a',-5,0]

animals = animal_dict(L)
for n,a in animals.items():
    print(f'key {n} with val {a}')
```

Manually loop over animal objects and access their data attr through getter methods

```
key 2 with val animal:None:2
key 5 with val animal:None:5
key 0 with val animal:None:0
```

YOU TRY IT!

- Write a function that meets this spec.

```
def make_animals(L1, L2):
    """ L1 is a list of ints and L2 is a list of str
        L1 and L2 have the same length
    Creates a list of Animals the same length as L1 and L2.
    An animal object at index i has the age and name
    corresponding to the same index in L1 and L2, respectively. """

#For example:
L1 = [2,5,1]
L2 = ["blobfish", "crazyant", "parafox"]
animals = make_animals(L1, L2)
print(animals)     # note this prints a list of animal objects
for i in animals:  # this loop prints the individual animals
    print(i)
```

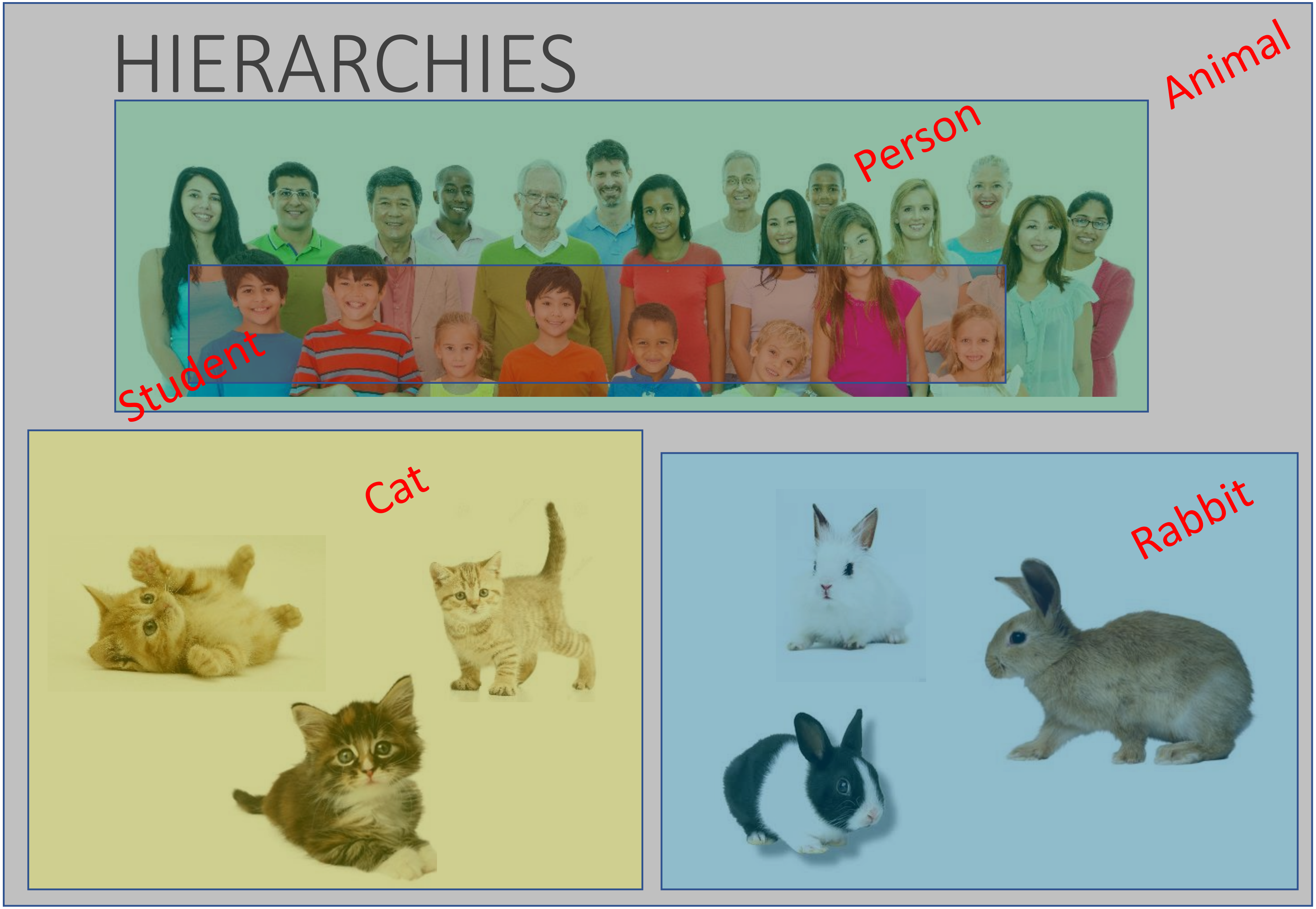
BIG IDEA

Access data attributes

(stuff defined by self.xxx)

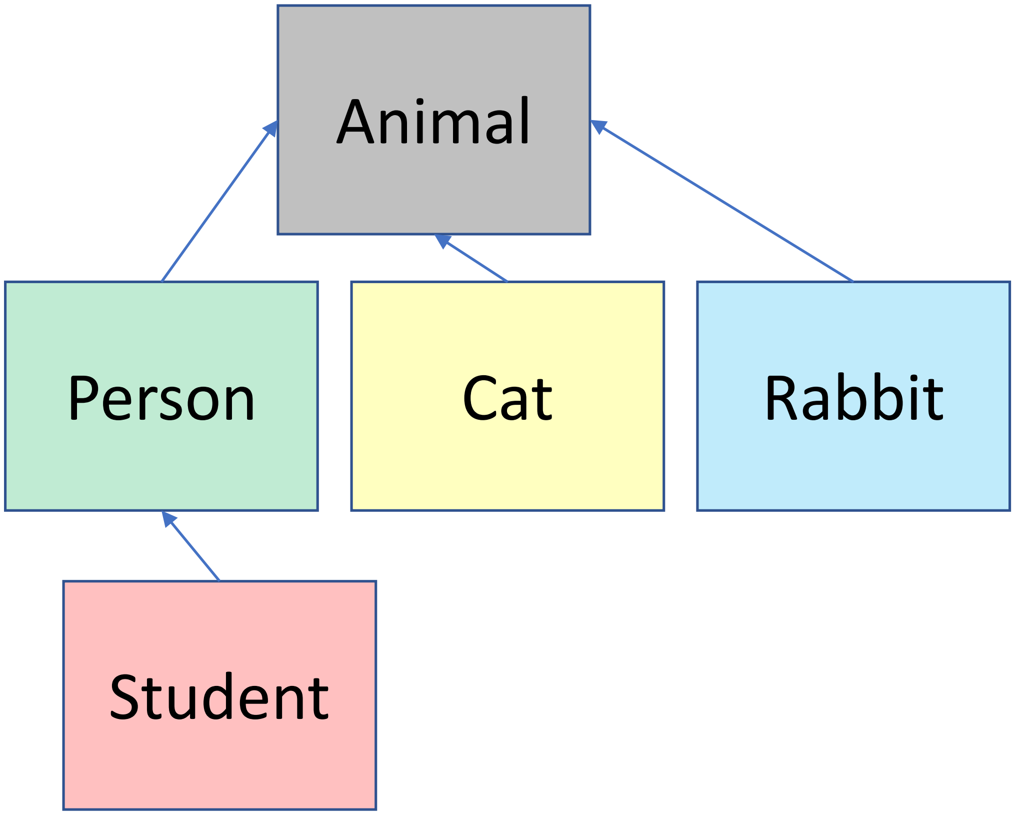
through methods – it’s better style.

HIERARCHIES



HIERARCHIES

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

SUBCLASS CAT

INHERITANCE: SUBCLASS

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

Add new functionality via speak method

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

Overrides __str__

- 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

WHICH METHOD TO USE?

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

SUBCLASS PERSON

```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 to run lines of code in Animal's init

Call `Animal`'s method

Add a new data attribute

New methods

Override `Animal`'s __str__ method

YOU TRY IT!

- Write a function according to this spec.

```
def make_pets(d):
    """ d is a dict mapping a Person obj to a Cat obj
    Prints, on each line, the name of a person, a colon, and the
    name of that person's cat """
    pass

p1 = Person("ana", 86)
p2 = Person("james", 7)
c1 = Cat(1)
c1.set_name("furball")
c2 = Cat(1)
c2.set_name("fluffsphere")

d = {p1:c1, p2:c2}
make_pets(d)  # prints ana:furball
              #        james:fluffsphere
```

BIG IDEA

A subclass can
use a parent’s attributes,
override a parent’s attributes, or
define new attributes.

Attributes are either data or methods.

SUBCLASS STUDENT

```python
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'm still zooming")
    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

Person __init__ takes care of all initializations

Adds new data

- I looked up how to use the `random` library in the python docs
- `random()` method gives back float in [0, 1)

SUBCLASS RABBIT

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

Shared class variable

instance variable

Access shared 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

RECALL THE `__init__` OF `Rabbit`

```
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
```

Shared across all instances

`Rabbit.tag` 2

`r1`

Age: 8
Parent1: None
Parent2: None
Rid: 1

`r1 = Rabbit(8)`

RECALL THE `__init__` OF `Rabbit`

```
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
```

Shared across all instances

```
r1 = Rabbit(8)
r2 = Rabbit(6)
```

Rabbit.tag

3

r1

Age: 8
Parent1: None
Parent2: None
Rid: 1

r2

Age: 6
Parent1: None
Parent2: None
Rid: 2

RECALL THE `__init__` OF `Rabbit`

```
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
```

Shared across all instances

```
r1 = Rabbit(8)
r2 = Rabbit(6)
r3 = Rabbit(10)
```

`Rabbit.tag` 4

`r1`
Age: 8
Parent1: None
Parent2: None
Rid: 1

`r2`
Age: 6
Parent1: None
Parent2: None
Rid: 2

`r3`
Age: 10
Parent1: None
Parent2: None
Rid: 3

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
```

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

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`**

RECALL THE `__init__` OF `Rabbit`

```
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
```

Shared across all instances

`Rabbit.tag` 5

```
r1 = Rabbit(8)
r2 = Rabbit(6)
r3 = Rabbit(10)
r4 = r1 + r2
```

Rabbit(0,r1, r2)

`r1`
Age: 8
Parent1: None
Parent2: None
Rid: 1

`r2`
Age: 6
Parent1: None
Parent2: None
Rid: 2

`r3`
Age: 10
Parent1: None
Parent2: None
Rid: 3

`r4`
Age: 0
Parent1: obj bound to r1
Parent2: obj bound to r2
Rid: 4

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.p1.rid == oth.p1.rid and self.p2.rid == oth.p2.rid)
    parents_opp = (self.p2.rid == oth.p1.rid and self.p1.rid == oth.p2.rid)
    return parents_same or parents_opp
```

Booleans checking r1+r2 or r2+r1

- Compare ids of parents since **ids are unique** (due to class var)
- Note you can't compare objects directly
  - For ex. with `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`

BIG IDEA

Class variables are shared between all instances.

If one instance changes it, it’s changed for every instance.

OBJECT ORIENTED PROGRAMMING

- Create your own **collections of data**
- **Organize** information
- **Division** of work
- Access information in a **consistent** manner

- Add **layers** of complexity
  - Hierarchies
  - Child classes inherit data and methods from parent classes
- Like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

ITERATORS

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from.

All these objects have a `iter()` method which is used to get an iterator:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

✓ 0.0s

```
apple
banana
cherry
```

ITERATORS

To create an object/class as an iterator you have to implement the methods `__iter__()` and `__next__()` to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called `__init__()`, which allows you to do some initializing when the object is being created.

The `__iter__()` method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The `__next__()` method also allows you to do operations, and must return the next item in the sequence.

```python
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

[2] ✓ 0.0s

```
1
2
3
4
5
```

ITERATORS

The example above would continue forever if you had enough next() statements, or if it was used in a `for` loop.

To prevent the iteration from going on forever, we can use the `StopIteration` statement.

In the `__next__()` method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

```python
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 4:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

[4] ✓ 0.0s

```
1
2
3
4
5
```

GENERATORS

A generator function is a special type of function that returns an iterator object. Instead of using return to send back a single value, generator functions use yield to produce a series of results over time. This allows the function to generate values and pause its execution after each yield, maintaining its state between iterations.

```python
def fun(max):
    cnt = 1
    while cnt <= max:
        yield cnt
        cnt += 1

ctr = fun(5)
for n in ctr:
    print(n)
```

Output

```
1
2
3
4
5
```

GENERATORS

Creating a generator in Python is as simple as defining a function with at least one yield statement. When called, this function doesn't return a single value; instead, it returns a generator object that supports the iterator protocol.

```
# A generator function that yields 1 for first time,
# 2 second time and 3 third time
def fun():
    yield 1
    yield 2
    yield 3

# Driver code to check above generator function
for val in fun():
    print(val)
```

Output

```
1
2
3
```

SCOPE

If you need to create a global variable, but are stuck in the local scope, you can use the `global` keyword.

The `global` keyword makes the variable global.

```
def myfunc():
  global x
  x = 300

myfunc()

print(x)
```

[5] ✓ 0.0s

```
300
```

SCOPE

To change the value of a global variable inside a function, refer to the variable by using the `global` keyword:

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

[6] ✓ 0.0s

```
... 200
```

SCOPE

The `nonlocal` keyword is used to work with variables inside nested functions.

The `nonlocal` keyword makes the variable belong to the outer function.

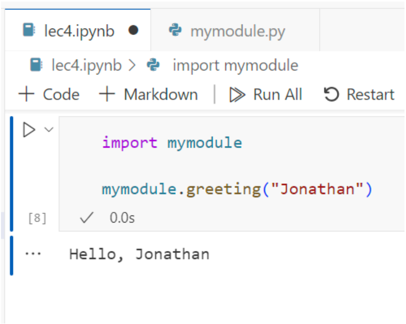
```python
def myfunc1():
  x = "Jane"
  def myfunc2():
    nonlocal x
    x = "hello"
  myfunc2()
  return x

print(myfunc1())
```

[7] ✓ 0.0s

```
hello
```

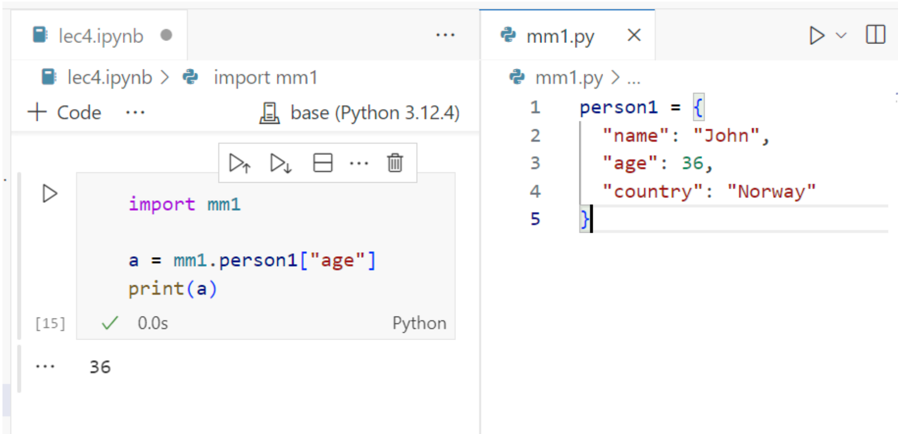
MODULES

To create a module just save the code you want in a file with the file extension `.py`:



MODULES

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):



MODULES

You can create an alias when you import a module, by using the `as` keyword

```
import mm1 as m

a = m.person1["age"]
print(a)
```

[17] ✓ 0.0s

```
36
```

MODULES

There are several built-in modules in Python, which you can import whenever you like.

```python
import platform

x = platform.system()
print(x)
```

[18] ✓ 0.0s

```
Windows
```

Using the dir() Function

There is a built-in function to list all the function names (or variable names) in a module. The `dir()` function

```
import platform

x = dir(platform)
print(x)
```

[19] ✓ 0.0s

```
['_Processor', '_WIN32_CLIENT_RELEASES', '_WIN32_SERVER_RELEASES',
```

DATETIME

A date in Python is not a data type of its own, but we can import a module named `datetime` to work with dates as date objects.

```python
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

[21] ✓ 0.0s

```
2025
Monday
```

DATETIME

To create a date, we can use the `datetime()` class (constructor) of the `datetime` module.

The `datetime()` class requires three parameters to create a date: year, month, day.

```python
import datetime

x = datetime.datetime(2020, 5, 17)

print(x)
```

[22] ✓ 0.0s

```
2020-05-17 00:00:00
```

DATETIME

The `datetime` object has a method for formatting date objects into readable strings.

The method is called `strftime()`, and takes one parameter, `format`, to specify the format of the returned string:

```
import datetime

x = datetime.datetime(2018, 6, 1)

print(x.strftime("%B"))
```

[23] ✓ 0.0s

```
June
```

Python math

The `min()` and `max()` functions can be used to find the lowest or highest value in an iterable

The `abs()` function returns the absolute (positive) value of the specified number

The `pow(x, y)` function returns the value of x to the power of y (x^y)

```python
x = min(5, 10, 25)
y = max(5, 10, 25)

print(x)
print(y)
```

[24] ✓ 0.0s

```
5
25
```

```python
x = abs(-7.25)

print(x)
```

[25] ✓ 0.0s

```
7.25
```

```python
x = pow(4, 3)

print(x)
```

[26] ✓ 0.0s

```
64
```

Python math

Python has also a built-in module called `math`, which extends the list of mathematical functions.

The `math.sqrt()` method for example, returns the square root of a number

```
import math

x = math.sqrt(64)

print(x)
```

[27] ✓ 0.0s

```
8.0
```

Python math

The `math.ceil()` method rounds a number upwards to its nearest integer, and the `math.floor()` method rounds a number downwards to its nearest integer, and returns the result

```python
import math

x = math.ceil(1.4)
y = math.floor(1.4)

print(x) # returns 2
print(y) # returns 1
```

[28] ✓ 0.0s

```
2
1
```

JSON

Python has a built-in package called `json`, which can be used to work with JSON data.

JSON is text, written with JavaScript object notation.

If you have a JSON string, you can parse it by using the `json.loads()` method.

```python
import json

# some JSON:
x =  '{ "name":"John", "age":30, "city":"New York"}'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])
```

[29] ✓ 0.0s

```
30
```

JSON

If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method.

```
import json

# a Python object (dict):
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)
```

[30] ✓ 0.0s

```
{"name": "John", "age": 30, "city": "New York"}
```

JSON

You can convert Python objects of the following types, into JSON strings:

- dict
- list
- tuple
- string
- int
- float
- True
- False
- None

```python
import json

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

[31] ✓ 0.0s

```
{"name": "John", "age": 30}
["apple", "bananas"]
["apple", "bananas"]
"hello"
42
31.76
true
false
null
```

JSON

When you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent:

Python	**JSON**
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

JSON

You can also define the separators, default value is (", ", ": "), which means using a comma and a space to separate each object, and a colon and a space to separate keys from values:

```python
import json

x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}

print(json.dumps(x, indent=4, separators=(". ", " = ")))
```

[33] ✓ 0.0s

```
{
    "name" = "John".
    "age" = 30.
    "married" = true.
    "divorced" = false.
    "children" = [
        "Ann".
        "Billy"
    ].
    "pets" = null.
```

JSON

The `json.dumps()` method has parameters to order the keys in the result

```
json.dumps(x, indent=4, sort_keys=True)
```

[34] ✓ 0.0s

```
'{\n    "age": 30,\n    "cars": [\n        {\n            "model": "BMW 230",\n            "mpg": 27.5\n        },\n        {\n            "model": "Fo
```

JSON

The JSON package in Python has a function called json.dumps() that helps in converting a dictionary to a JSON object. It takes two parameters:

- **dictionary** – the name of a dictionary which should be converted to a JSON object.
- **indent** – defines the number of units for indentation

```python
import json

# Data to be written
dictionary = {
    "name": "sathiyajith",
    "rollno": 56,
    "cgpa": 8.6,
    "phonenumber": "9976770500"
}

# Serializing json
json_object = json.dumps(dictionary, indent=4)

# Writing to sample.json
with open("sample.json", "w") as outfile:
    outfile.write(json_object)
```

Output:

```
{
    "name": "sathiyajith",
    "rollno": 56,
    "cgpa": 8.6,
    "phonenumber": "9976770500"
}
```

JSON

The JSON package has json.load() function that loads the JSON content from a JSON file into a dictionary.
It takes one parameter:

- **File pointer:** A file pointer that points to a JSON file.

```
import json

with open("sample-data.json","r") as openfile:
    data = json.load(openfile)

print(data)
```

[11]

{'totalCount': '400', 'imdata': [{'l1PhysIf': {'attributes': {'adminSt