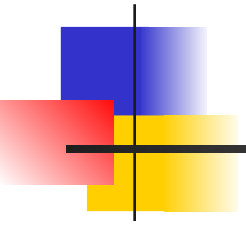


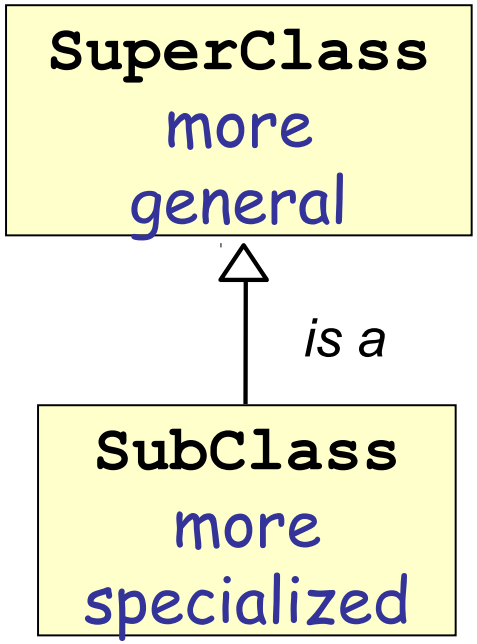
Introduction to Inheritance

James Brucker

What is Inheritance?

One class incorporates all the attributes and behavior from another class -- it *inherits* these attributes and behavior.

- A subclass inherits **all** the attributes and behavior of the superclass.
- It can directly access the public & protected members of the superclass.
- Subclass can *redefine* some inherited behavior, or add new attributes and behavior.



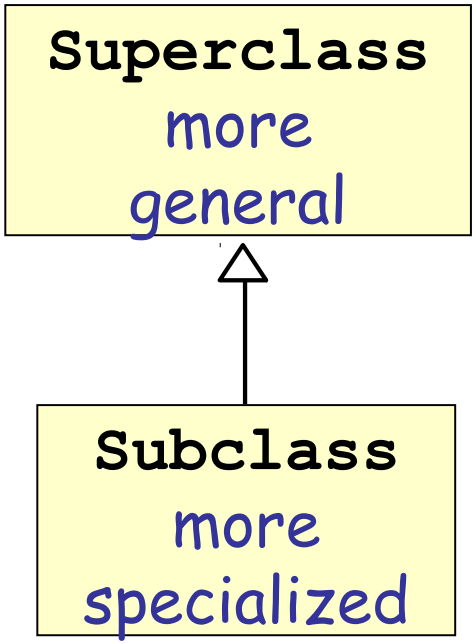
UML for inheritance

Terminology

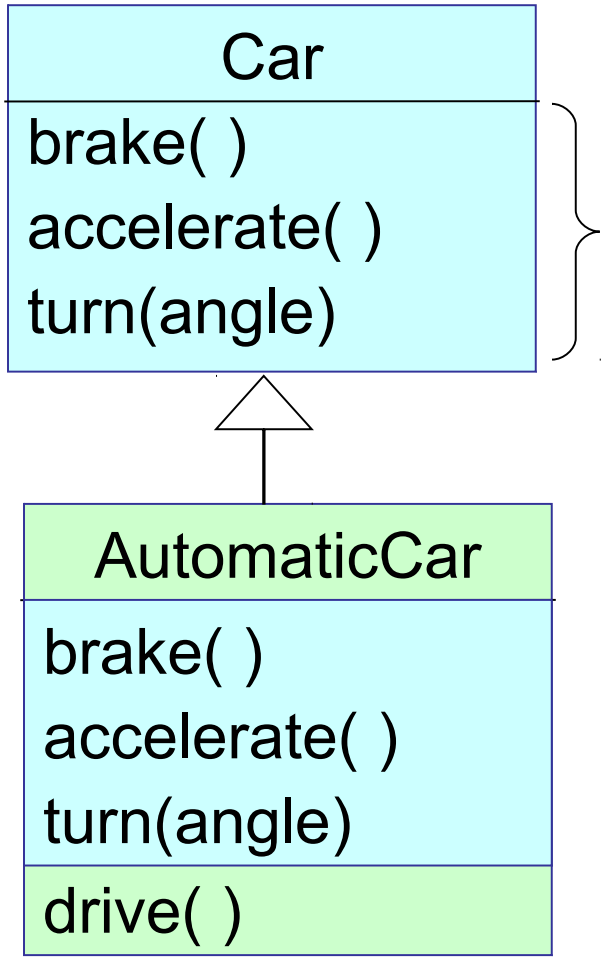
Different names are used for inheritance relationships.

They mean *the same thing*.

Superclass	**Subclass**
parent class	child class
base class	derived class



"Specializing" or "Extending" a Type



Consider a basic Car.

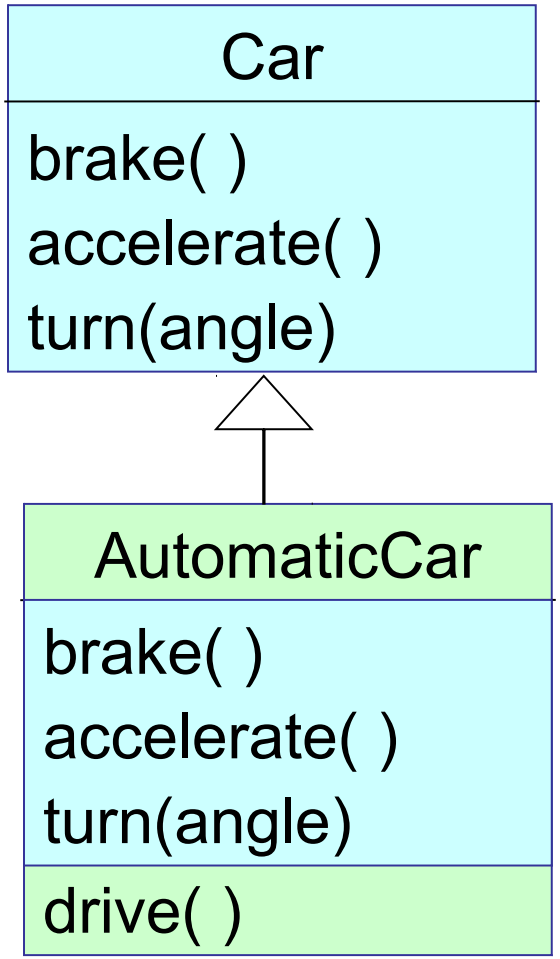
the behavior of a Car

An AutomaticCar is a *special kind* of Car with automatic transmission.

AutomaticCar can do anything a Car can do.

It also adds *extra behavior*.

Benefit of Extending a Type



Extension has some benefits:

Benefit to user

If you can drive a basic Car,
you can drive an Automatic Car.
It works (almost) the same.

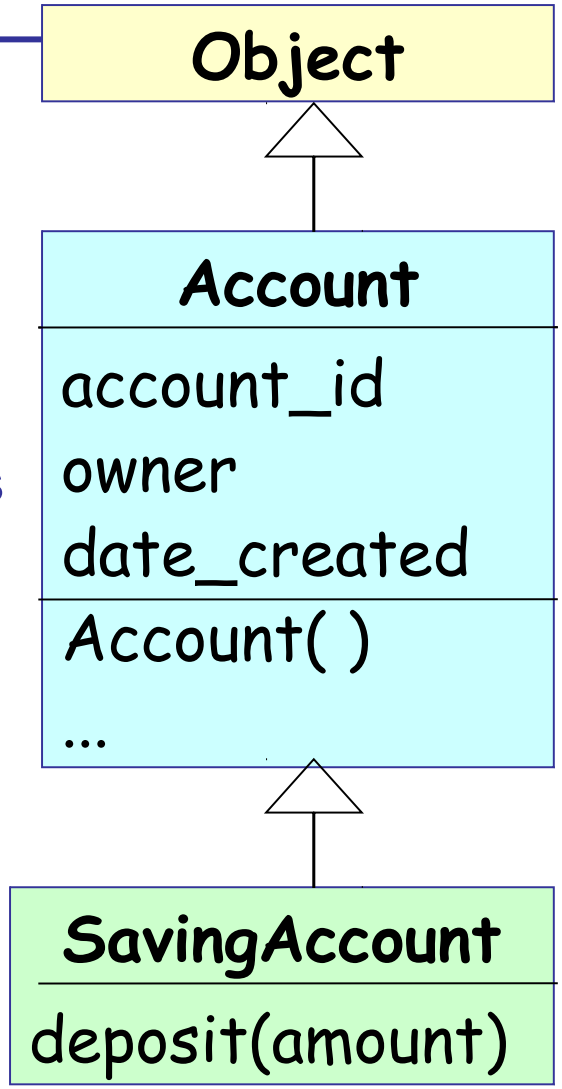
Benefit to producer (programmer)

You can *reuse* the behavior from Car
to create AutomaticCar.
Just add automatic "drive".

What do you *inherit*?

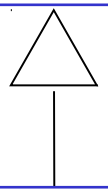
A subclass *inherits from its parent classes:*

- ✓ attributes
- ✓ methods - even private ones.
- ✓ cannot directly *access* "**`private`**" members of parent class, but they are inherited



Syntax for Inheritance

```
class SuperClass:
    ...
```

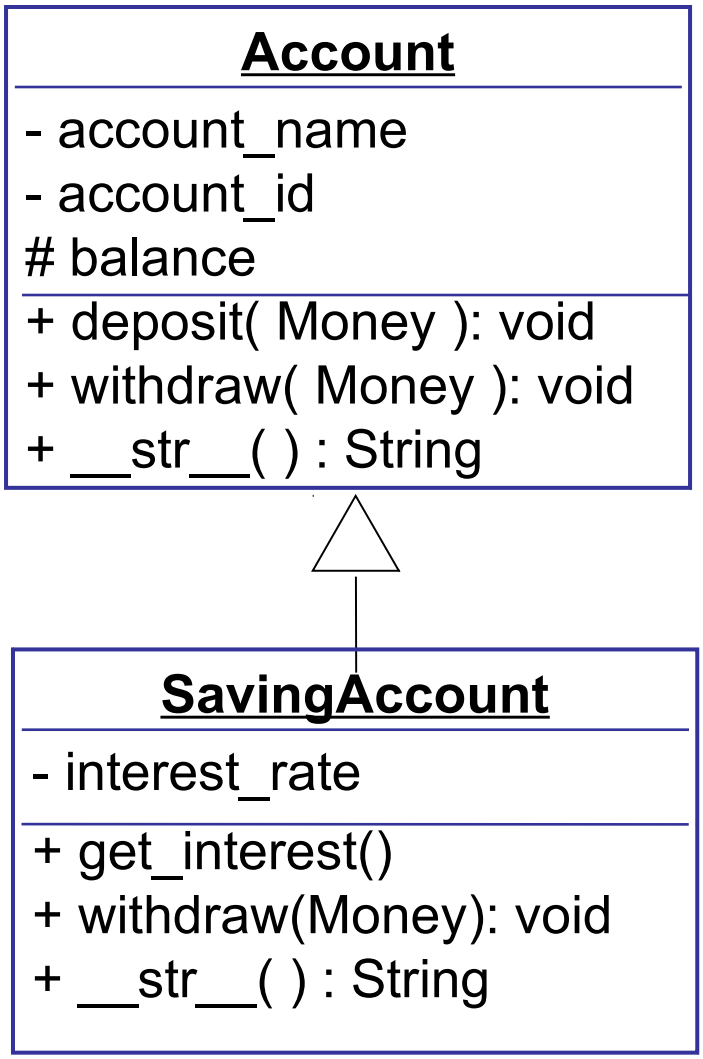


```
class SubClass(SuperClass):
    ...
```

Interpretation of Inheritance (1)

Superclass defines basic behavior and attributes.

Subclass specializes some behavior, may add new behavior (get_interest).



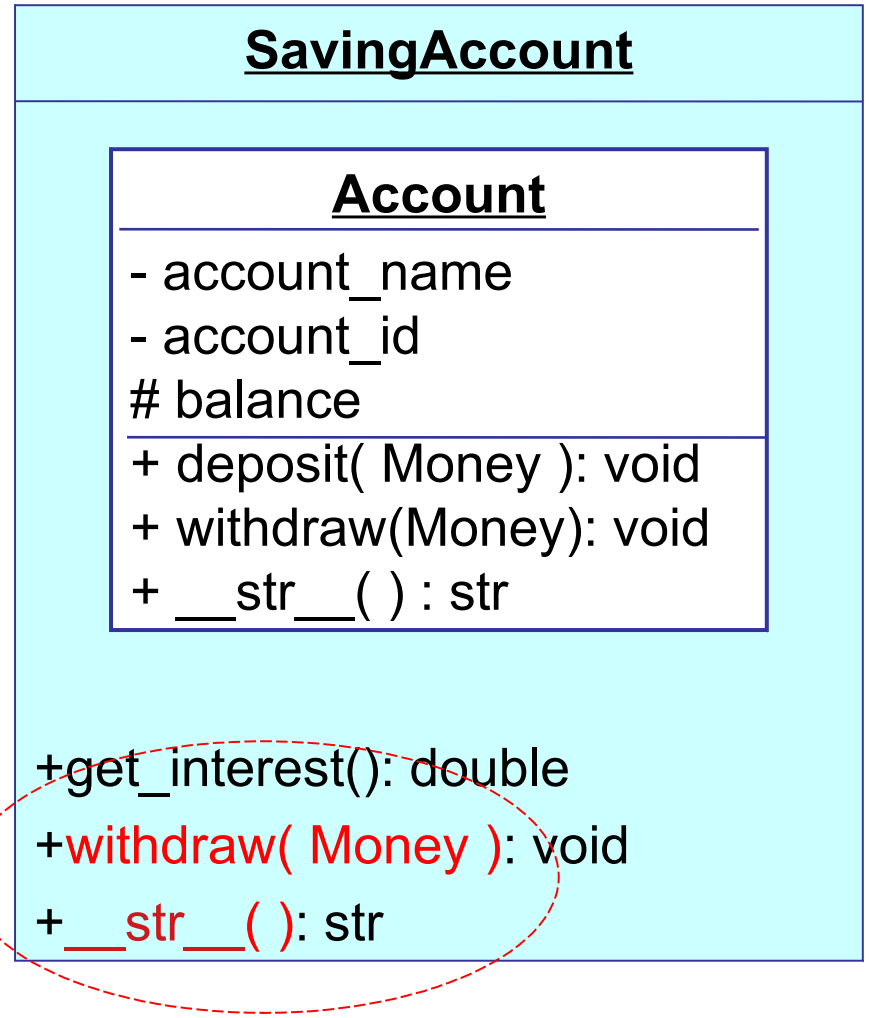
Use of Inheritance

A subclass can...

- **add new** behavior and attributes (extension)
- **redefine** existing behavior (specialize)

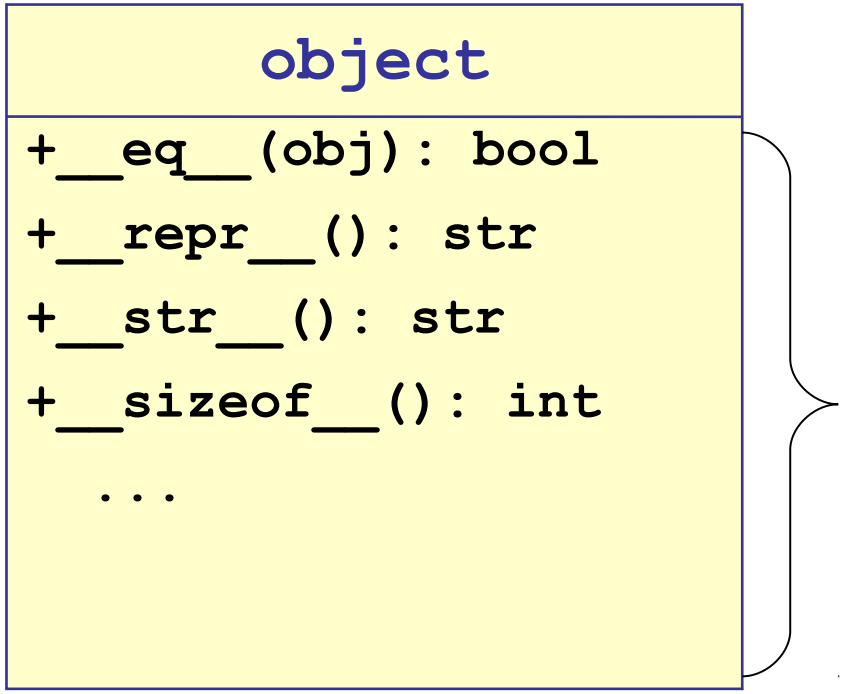
Subclass can override methods to specialize its behavior.

SavingAccount overrides withdraw and __str__.



object: the Universal Superclass

- All Python classes are subclasses of **`object`**.
- You **don't** write "`class MyClass(object)`".
- `object` defines basic methods for all classes:



Every class is guaranteed to have these methods.

Most of them do nothing or raise Exception.

Calling a Superclass Method

A subclass can call a method of its superclass.

Typically, a subclass constructor *should* call the parent constructor *first*.

```
class SavingAccount(Account):


    def __init__(self, acct_number, acct_name, int_rate):
        # acct number and acct name are attributes
        # of all Accounts.
        super().__init__(acct_number, acct_name)
        # interest is specific to SavingsAccount
        self.interest_rate = int_rate
```

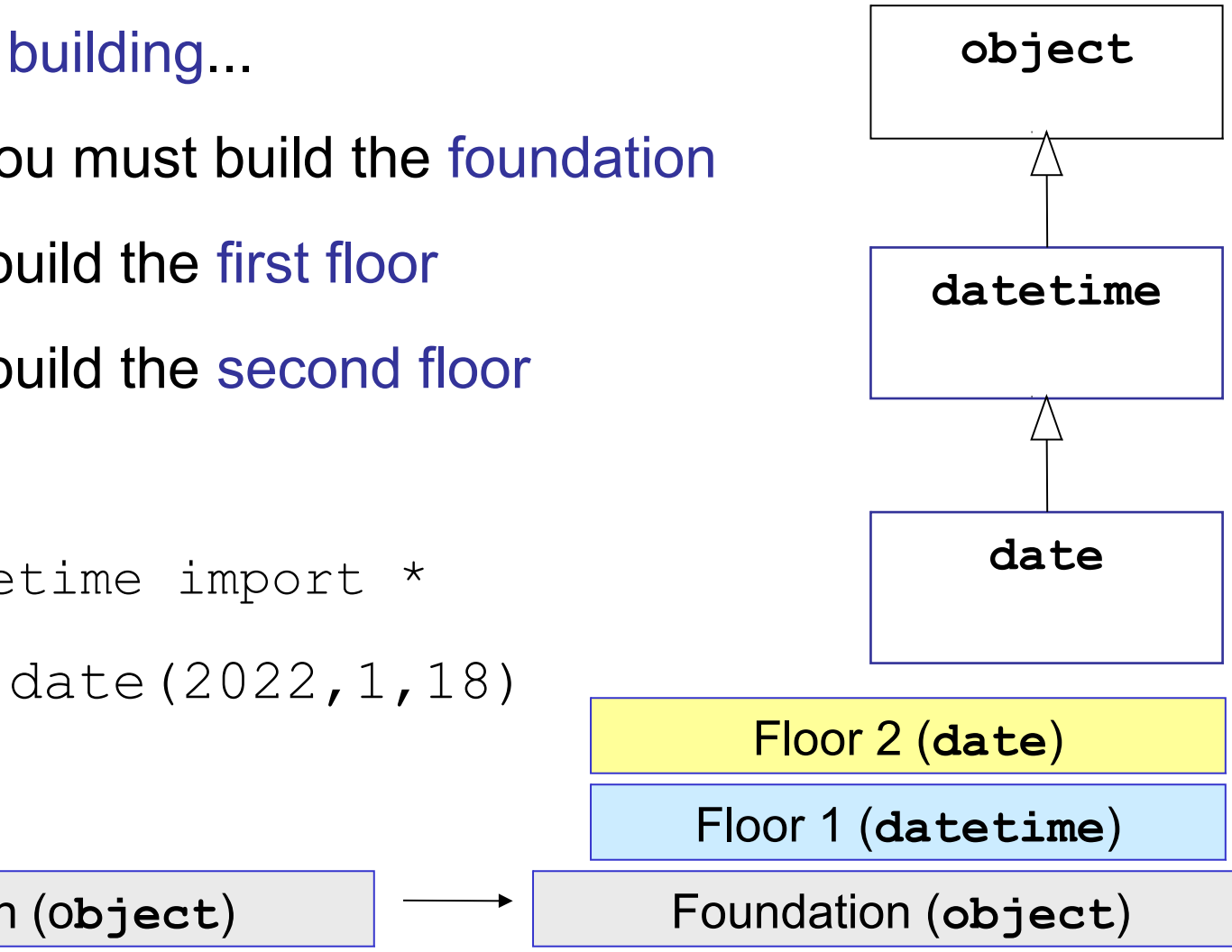
Constructors and Inheritance

To **build** a building...

- first you must build the foundation
- then build the first floor
- then build the second floor
- *etc*.

```
from datetime import *

today = date(2022,1,18)
```



Try It!

Write a Person class and Student subclass.

- Person has a name.

  str(person) returns:

  `"Person person-name"`

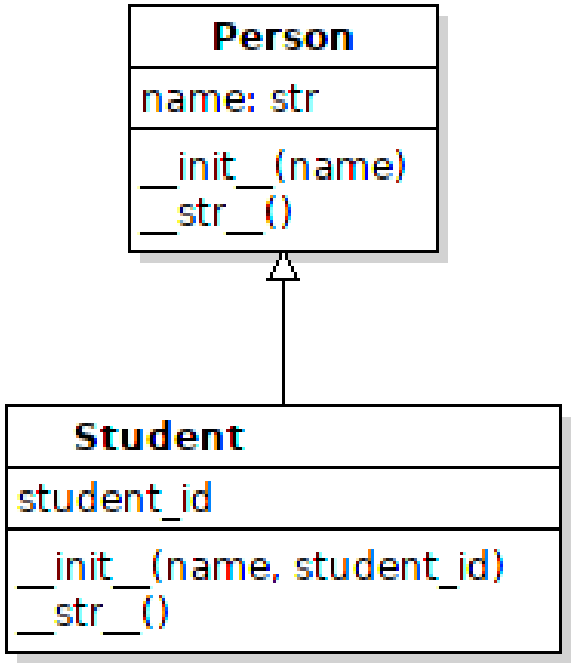
- Student has a name and student_id

  str(student) returns:

  `"Student name, id student-id"`

- main block creates a Student

  and prints it.



Attribute Visibility

In Student you should have:

```
class Student(Person):
    def __init__(self, name: str, id: int):
        super().__init__(name)
        self.student_id = id

    def __str__(self):
        return f"Student {self.name},
                                id {self.student_id}"
```

Protected Attributes

1. In **Person** rename the attribute to: **_name**

```
class Person:
    def __init__(self, name):
        self._name = name

class Student(Person):
   def __str__(self):
        return f"Student {self._name},
                                id {self.student_id}"
```

Run the code.

Does it still work?

Private Attributes

1. In **Person** rename the attribute to: **__name**

```
class Person:
    def __init__(self, name):
        self.__name = name

class Student(Person):
   def __str__(self):
        return f"Student {self.__name},
                                  id {self.student_id}"
```

Run the code.

Does it still work?

Private Attribute + Public Property

When an atrribute is **private** (**__name**),
subclasses cannot directly access the value.
This is **good**!
Write a property for read-only access.

```
class Person:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name

In Student invoke the property using:
    self.name              # not super().name
```

When to write `super()`?

You only need to write `super()` if a class has a method with **same name** as a superclass method, and you explicitly want to call the superclass method.

```
class Person:
    def __str__(self):     # in Person & Student
        return f"{self.__name}"

    def birthday(self):    # only in Person
        return self._birthday

class Student(Person):
    def __str__(self):
       return super().__str__() \
                     +" born on " + self.birthday()
```

When to write super( )?

In the Student constructor, we want to call the superclass constructor. Since both Student and Person have a constructor (__init__), we must write super().__init__(...)

```
class Student(Person):
    def __init__(self, name, id):
        super().__init__(name)
        self.id = id
```

What is the class hierarchy for `ValueError`?

Hint: >>> help(ValueError)
>>> help(superclass_of_ValueError)

ValueError is a subclass of Exception
Exception is a subclass of BaseException
BaseException is a subclass of object

```
class Student(Person, list)   # try to avoid this
s = Student("Joe", 1234567890)
s.foo( )
```

When to use Inheritance?

- You have a "hierarchy" of related types, where some behavior is the same and some is different (specialization).
- Use inheritance to "factor out" common behavior (put it in the superclass).

Misuse of Inheritance

1. Subclass is not truly a subtype of the superclass.

   Stack is not a subclass of list.

   Both Stack and list are ordered collections of objects, but a list has behavior a Stack should not have. You can add/remove items anywhere in a list, but only at the top of Stack.

2. Subclass is really an instance:

   JoeBiden is a President,

   but JoeBiden is not a subclass of President.
   He is an *instance* of President.

Misuse of Inheritance

3. Subclass does not override any behavior or add any new behavior.

Classes are about *behavior*. If both the Parent and Child class do the same thing, there is no reason for the Child.

4. The "subclass" can change.
Model this as a role (attribute) not a subclass.

A `ParttimeStudent` is a kind of `Student`.
`ParttimeStudent` specializes some student behavior.

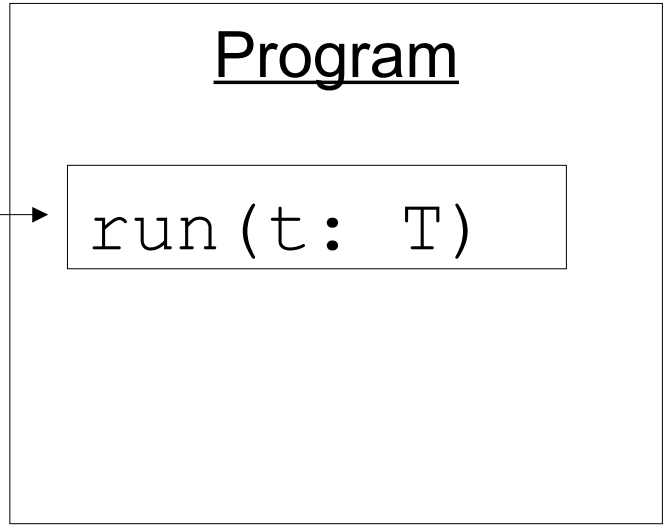
But, a ParttimeStudent can change his status to a regular (full time) Student.

Liskov Substitution Principle

"If S is a subtype of T, then in any program that uses an object of type T we can replace the T-object with an object of type S, without altering any expected properties of the program (e.g. the program still works correctly)."

```
t = S()

program.run( t )
```



Applying Liskov Substitution Principle

```
greet(person: Person)
"""Greet a Person.  If today is his birthday then say Happy Birthday,
    otherwise just say a usual greeting."""
```

Student also has a name and a birthday, and accessor methods for both, just like Person.

We should be able to write:

```
greet(student)
```

the program will still work.

So LSP is satisfied in this case.

References

Inheritance in the Official Python Tutorial (section 9.5)

This is easy to understand and a definitive guide.

https://docs.python.org/3.8/tutorial/classes.html#inheritance

Inheritance and Composition on realpython.com

Explains both inheritance and composition, and when to use each one.

https://realpython.com/inheritance-composition-python/

Inheritance chapter in *Think Python* online book

An OK (not great) intro to inheritance. The deck of cards example is not very good, in my opinion.