4. előadás

Python és az OO 3.

Programozás (2) előadás

2022. Október 3.

Halász Gábor

Debreceni Egyetem

Általános tudnivalók

Ajánlott irodalom:

- Nyékyné G. Judit (szerk): Programozási nyelvek, Kiskapu, 2003.
- Juhász, István: Magas szintű programozási nyelvek 2, elektronikus egyetemi jegyzet, 2009
- Tarczali, Tünde: UML diagramok a gyakorlatban, Typotex Kiadó, 2011.
- Angster, Erzsébet: Objektumorientált tervezés és programozás: JAVA, 4KÖR Bt., 2002, ISBN: 9632165136
- Bird, S., Klein, E., Loper, E.: Natural Language Processing with Python, O'Reilly Media, 2009

Félév teljesítésének feltételei: jelenlét + 2 gyakorlati + 1 elméleti ZH

Érdemjegy: $1 < 60\% \leq 2 < 70\% \leq 3 < 80\% \leq 4 < 90\% \leq 5$

További részletek: `https://elearning.unideb.hu/`

Objektumorientáltság és a Python

OOP principles (again)

encapsulation: hiding design details to make the program clearer and more easily modified later

modularity: the ability to make objects “stand alone” so they can be reused (our modules). Like the math module

inheritance: create a new object by inheriting (like father to son) many object characteristics while creating or over-riding for this object

polymorphism: (hard) Allow one message to be sent to any object and have it respond appropriately based on the type of object it is.

We are still at encapsulation

We said that encapsulation:

- hid details of the implementation so that the program was easier to read and write
- modularity, make an object so that it can be reused in other contexts
- providing an interface (the methods) that are the approved way to deal with the class

One more aspect

- A new aspect we should have is consistency
  Remember Rule 9: *Do the right thing*
- A new class should be consistent with the rules of the language.
- It should respond to standard messages, it should behave properly with typical functions (assuming the type allows that kind of call).

An example

Consider a Rational number class. It should respond to:

- construction
- printing
- arithmetic ops (+, -, *, /)
- comparison ops (<, >, <=, >=)

example program

```
# get our rational number class named frac_class
>>> from frac_class import *
>>> r1 = Rational(1,2)      # create the fraction 1/2
>>> r2 = Rational(3,2)      # create the fraction 3/2
>>> r3 = Rational(3)        # default denominator is 1, so really creating 3/1
>>> r_sum = r1 + r2         # use "+" in a familiar way
>>> print(r_sum)            # use "print" in a familiar way
4/2
>>> r_sum                   # display value in session in a familiar way
4/2
>>> if r1 == r1:            # use equality check "==" in a familiar way
...     print('equal')
... else:
...     print('not equal')
...
equal
>>> print(r3 - r2)           # combine arithmetic and printing in a familiar way
3/2
```

just like any other number

- by building the class properly, we can make a new instance of Rational look like any other number syntactically.
- the instance responds to all the normal function calls
- because it is properly encapsulated, it is much easier to use

But how can that work?

Two parts:

- Python can distinguish which operator to use based on types
- Python provides more standard methods that represent the action of standard functions in the language
  - by defining them in our class, Python will call them in the "right way"

More on type

- As we have mentioned, a class is essentially a new type
- when we make an instance of a class, we have made an object of a particular type
- 1.36 is a float
- after

  `my_instance = MyClass()`,

  `my_instance` is a type `MyClass`

Introspection

- Python does not have a type associated with any variable, since each variable is allowed to reference any object
- however, we can query any variable as to what type it presently references
- this is often called introspection. That is, while the program is running we can determine the type a variable references

Python introspection ops

- `type(variable)`
  - returns its type as an object
- `isinstance(variable,type)`
  - returns a boolean indicating if the variable is of that type

```python
def special_sum(a,b):
    ''' sum two ints or convert params to ints
    and add. return 0 if conversion fails '''
    if type(a)==int and type(b)==int:
        result = a + b
    else:
        try:
            result = int(a) + int(b)
        except ValueError:
            result = 0
    return result
```

Operator Overloading

So what does `var1+var2` mean?

The answer:

- it depends
- What it depends on is the type. The + operation has two operands. What are their types?
- Python uses introspection to find the type and then select the correct operator

We’ve seen this before

What does `var1+var2` do?

- with two strings, we get concatenation
- with two integers, we get addition
- with an integer and a string we get:

```
Traceback (most recent call last):
File "<pyshell#9>", line 1, in
<module>
    1+'a'
TypeError:  unsupported operand
type(s) for +:  'int' and 'str'
```

Operator overloading

- the plus operator is **overloaded**
- that is, the operator can do/mean different things (have multiple/overloaded meanings) depending on the types involved
- if python does not recognize the operation and that combination of types, you get an error

Python overload ops

- Python provides a set of operators that can be overloaded. You can't overload all the operators, but you can many
- Like all the special class operations, they use the two underlines before and after They come in three general classes:
    - numeric type operations (+,-,<,>,print etc.)
    - container operations ([ ], iterate,len, etc.)
    - general operations (printing, construction)

Math-like Operators		
Expression	**Method name**	**Description**
x + y	`__add__()`	Addition
x − y	`__sub__()`	Subtraction
x * y	`__mul__()`	Multiplication
x / y	`__div__()`	Division
x == y	`__eq__()`	Equality
x > y	`__gt__()`	Greater than
x >= y	`__ge__()`	Greater than or equal
x < y	`__lt__()`	Less than
x <= y	`__le__()`	Less than or equal
x != y	`__ne__()`	Not equal
Sequence Operators		
len(x)	`__len__()`	Length of the sequence
x in y	`__contains__()`	Does the sequence y contain x?
x[key]	`__getitem__()`	Access element *key* of sequence x
x[key]=y	`__setitem__()`	Set element *key* of sequence x to value y
General Class Operations		
x=myClass()	`__init__()`	Constructor
print (x), str(x)	`__str__()`	Convert to a readable string
	`__repr__()`	Print a Representation of x
	`__del__()`	Finalizer, called when x is garbage collected

```python
class MyClass(object):
        def __init__(self, param1=0):
                ''' constructor, sets attribute value to
                param1, default is 0'''
                print('in constructor')
                self.value = param1

        def __str__(self):
                ''' Convert val attribute to string.'''
                print('in str')
                return 'Val is: {}'.format(str(self.value))

        def __add__(self,param2):
                ''' Perform addition with param2, a MyClass instance.
                Return a new MyClass instance with sum as value attribute'''
                print('in add')
                result = self.value + param2.value
                return MyClass(result)
```

how does v1+v2 map to __add__

`v1 + v2`

is turned, by Python, into

`v1.__add__(v2)`

- These are **exactly equivalent expressions.** It means that the first variable calls the `__add__` method with the second variable passed as an argument
- `v1` is bound to `self`, `v2` bound to `param2`

Calling __str__

- When does the __str__ method get called? Whenever a string representation of the instance is required:
  - directly, by saying `str(my_instance)`
  - indirectly, calling `print(my_instance)`

Simple Rational Number class

Simple Rational Number class

- a Rational is represented by two integers, the numerator and the denominator
- we can apply many of the numeric operators to Rational

```python
class Rational(object):
    """ Rational with numerator and denominator. Denominator
    parameter defaults to 1"""
    def __init__(self,numer,denom=1):
        print('in constructor')
        self.numer = numer
        self.denom = denom

    def __str__(self):
        """ String representation for printing"""
        print('in str')
        return str(self.numer)+'/'+str(self.denom)

    def __repr__(self):
        """ Used in interpreter. Call __str__ for now """
        print('in repr')
        return self.__str__()
```

`__str__` VS `__repr__`

- `__repr__` is what the interpreter will call when you type an instance
  - potentially, the representation of the instance, something you can recreate an instance from.
- `__str__` is a conversion of the instance to a string.
  - Often we define `__str__`, have `__repr__` call `__str__` – note the call: `self.__str__()`

the __init__ method

- each instance gets an attribute `numer` and `denom` to represent the numerator and denominator of that instance's values

provide addition

Remember how we add fractions:

- if the denominator is the same, add the numerators
- if not, find a new common denominator that each denominator divides without remainder.
- modify the numerators and add

the `lcm` and `gcd`

- the least common multiple (`lcm`) finds the smallest number that each denominator divides without remainder
- the greatest common divisor (`gcd`) finds the largest number two numbers can divide into without remainder

LCM in terms of GCD

$$\mathrm{LCM}(a, b) = \frac{a * b}{\mathrm{GCD}(a, b)}$$

OK, how to find the `gcd` then?

GCD and Euclid

- One of the earliest algorithms recorded was the GCD by Euclid in his book Elements around 300 B.C.
  - He originally defined it in terms of geometry but the result is the same

The Algorithm

- `GCD(a,b)`

1. If one of the numbers is `0`, return the other and `halt`
2. Otherwise, find the integer `remainder` of the larger number divided by the `smaller_number`
3. Reapply `GCD(a,b)` with `a = smaller_number` and `b = remainder` from step 2)

```python
def gcd(bigger, smaller):
    """Calculate the greatest common divisor of two positive integers."""
    if not bigger > smaller:        # swap if necessary so bigger > smaller
        bigger, smaller = smaller, bigger
    while smaller != 0:                     # 1. if smaller == 0, halt
        remainder = bigger % smaller        # 2. find remainder
        print('calculation, big:{}, small:{}, rem:{}'.\
              format(bigger, smaller, remainder)) # debugging
        bigger, smaller = smaller, remainder  # 3. reapply
    return bigger
```


```python
def gcd(bigger, smaller):
    """Calculate the greatest common divisor of two positive integers."""
    if not bigger > smaller:        # swap if necessary so bigger > smaller
        bigger, smaller = smaller, bigger
    while smaller != 0:                     # 1. if smaller == 0, halt
        remainder = bigger % smaller        # 2. find remainder
        print('calculation, big:{}, small:{}, rem:{}'.\
              format(bigger, smaller, remainder)) # debugging
        bigger, smaller = smaller, remainder  # 3. reapply
    return bigger
```

__add__ and __sub__

```
    def __add__(self, param_Rational):
        """ Add two Rationals """
        print('in add')
        # find a common denominator (lcm)
        the_lcm = lcm(self.denom, param_Rational.denom)
        # multiply each by the lcm, then add
        numerator_sum = (the_lcm * self.numer/self.denom) + \
                        (the_lcm * param_Rational.numer/param_Rational.denom)
        return Rational(int(numerator_sum), the_lcm)

    def __sub__(self, param_Rational):
        """ Subtract two Rationals """
        print('in sub')
        # subtraction is the same but with '-' instead of '+'
        the_lcm = lcm(self.denom, param_Rational.denom)
        numerator_diff = (the_lcm * self.numer/self.denom) - \
                         (the_lcm * param_Rational.numer/param_Rational.denom)
        return Rational(int(numerator_diff), the_lcm)
```

Equality

- The equality method is `__eq__`
- It is invoked with the `==` operator
  - `1/2 == 1/2` is equivalent to
  - `1/2.__eq__(1/2)`
- It should be able to deal with non-reduced fractions:
  - `1/2 == 1/2` is `True` so is
  - `2/4 == 3/6`

```
def reduce_rational(self):
    """ Return the reduced fractional value as a Rational"""
    print('in reduce')
    # find the gcd and then divide numerator and denominator by gcd
    the_gcd = gcd(self.numer,self.denom)
    return Rational(self.numer//the_gcd, self.denom//the_gcd)

def __eq__(self,param_Rational):
    """ Compare two Rationals for equality, return Boolean"""
    print('in eq')
    # reduce both; then check that numerators and denominators are equal
    reduced_self = self.reduce_rational()
    reduced_param = param_Rational.reduce_rational()
    return reduced_self.numer == reduced_param.numer and\
        reduced_self.denom == reduced_param.denom
```

Fitting in

- What is amazing about the traces of these methods is how many of them are called in service of the overall goal.
- All we did was provide the basic pieces and Python orchestrates how they all fit together!
- Rule 9 rules!

What doesn't work

So r1+r2, but what about

- We said the add we defined would work for two rationals, but what about?
  `r1 + 1` *# Rational plus an integer*
  `1 + r1` *# commutativity*
- Neither works right now. How to fix?

r1 + 1

- What's the problem?
  - add expects another rational number as the second argument.
  - Python used to have a coercion operator, but that is deprecated
    - coerce: force conversion to another type
    - deprecate: 'disapproval', an approach that is no longer supported
  - Our constructor would support conversion of an int to a Rational, how/where to do this?

Introspection in `__add__`

- the `add` operator is going to have to check the types of the parameter and then decide what should be done
- if the type is an integer, convert it. If it is a Rational, do what we did before. Anything else that is to be allowed needs to be checked

```python
def __add__(self, param):
    """ Add two Rationals. Allows int as a parameter"""
    print('in add')
    if type(param) == int:  # convert ints to Rationals
        param = Rational(param)
    if type(param) == Rational:
        # find a common denominator (lcm)
        the_lcm = lcm(self.denom, param.denom)
        # multiply each by the lcm, then add
        numerator_sum = (the_lcm * self.numer/self.denom) + \
            (the_lcm * param.numer/param.denom)
        return Rational(int(numerator_sum),the_lcm)
    else:
        print('wrong type')  # problem: some type we cannot handle
        raise(TypeError)
```

what about 1 + r1

- What's the problem
  - mapping is wrong
  - `1 + r1` maps to `1.__add__(r1)`
  - no such method for integers
    (and besides, it would be a real pain to have to add a new method to every type we want to include)
  - user should expect that this should work.
    Addition is commutative!

radd method

- Python allows the definition of an `__radd__` method
- The `__radd__` method is called when the `__add__` method fails because of a type mismatch
- `__radd__` reverses the two arguments in the call

`__radd__` VS `__add__`

- `1 + r1`

  try `1.__add__(r1)`, *failure*

  look for an `__radd__` *if it exists, remap*

- `1 + r1`

  `r1.__radd__(1)`

radd

- essentially, all we need `__radd__` to do is remap the parameters.
- after that, it is just add all over again, so we call `__add__` directly
- means we only have to maintain `__add__` if any changes are required

```
def __radd__(self,f):
    return self.__add__(f)
```

Inheritance

Class-Instance relations

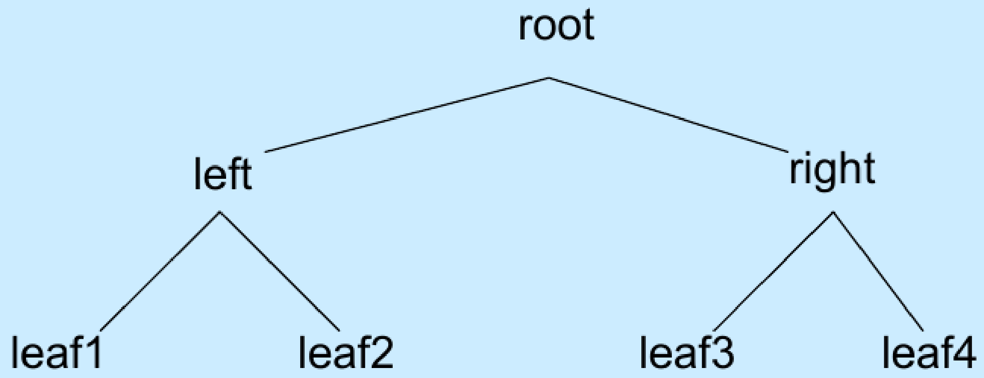
- Remember the relationship between a class and its instances
  - a class can have many instances, each made initially from the constructor of the class
  - the methods an instance can call are initially shared by all instances of a class

Class-Class relations

- Classes can also have a separate relationship with other classes
- the relationships forms a hierarchy
  - **hierarchy**: A body of persons or things ranked in grades, orders or classes, one above another

computer science 'trees'

- the hierarchy forms what is called a tree in computer science. Odd 'tree' though



Classes related by a hierarchy

- when we create a class, which is itself another object, we can state how it is related to other classes
- the relationship we can indicate is the class that is ’above’ it in the hierarchy

class statement

```
class MyClass (SuperClass):
    pass
```

`SuperClass`: name of the class above `MyClass` in the hierarchy

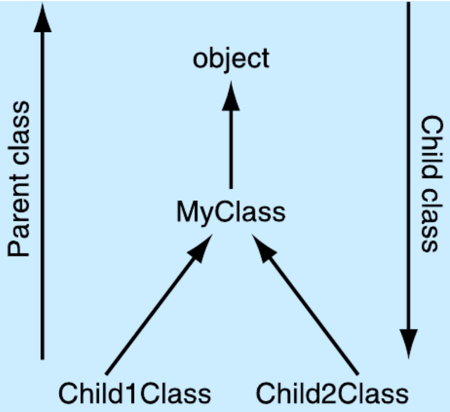
- The top class in Python is called `object`.
- it is predefined by Python, always exists
- use `object` when you have no superclass

A symple class hierarchy

```
class MyClass (object):
   pass


class Child1Class (MyClass):
   pass


class Child2Class (MyClass):
   pass
```



```python
class MyClass (object):
        ''' parent is object'''
        pass

class MyChildClass (MyClass):
        ''' parent is MyClass '''
        pass

my_child_instance = MyChildClass()
my_class_instance = MyClass()

print(MyChildClass.__bases__)          # the parent class
print(MyClass.__bases__)               # ditto
print(object.__bases__)                # ditto

print(my_child_instance.__class__)     # class from which the instance came
print(type(my_child_instance))         # same question, asked via function
```

is-a, super and sub class

- the class hierarchy imposes an **is-a** relationship between classes
  - MyChildClass is-a (or is a kind of) MyClass
  - MyClass is-a (or is a kind of) object
  - object has as a subclass MyClass
  - MyChildClass has as a superclass MyClass

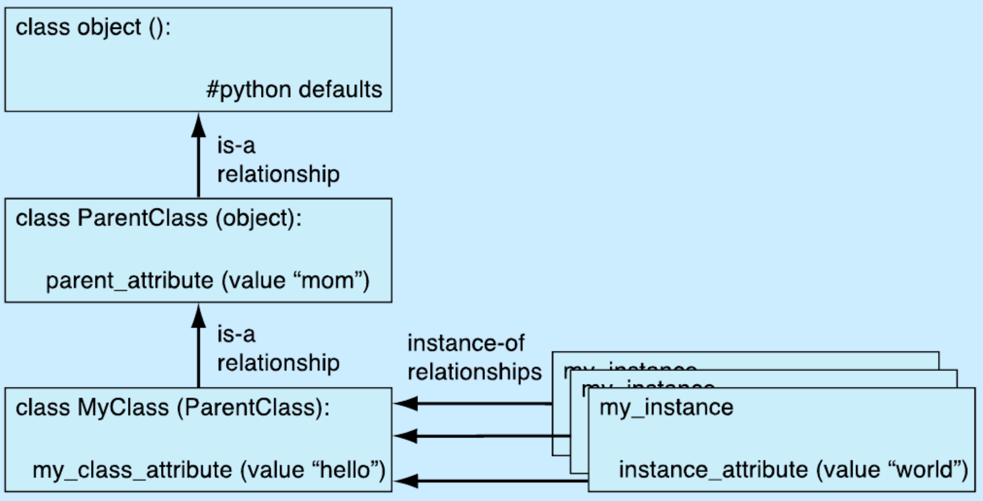
um, so what?

- the hope of such an arrangement is the saving/re-use of code
- superclass code contains general code that is applicable to many subclasses
- subclass uses superclass code (via sharing) but specializes code for itself when necessary

Scope for objects, the full story

1. Look in the object for the attribute
2. If not in the object, look to the object's class for the attribute
3. If not in the object's class, look up the hierarchy of that class for the attribute
4. If you hit object, then the attribute does not exist

The players in the „find the attribute" game



Inheritance is powerful but also can be complicated

- many powerful aspects of OOP are revealed through uses of inheritance
- However, some of that is a bit detailed and hard to work with. Definitely worth checking out but a bit beyond us and our first class

builtins are objects too

- One nice way, easy way, to use inheritance is to note that all the builtin types are objects also
- thus you can inherit the properties of builtin types then modify how they get used in your subclass
- you can also use any of the types you pull in as modules

specializing a method

- One technical detail. Normal method calls are called **bound methods**. Bound methods have an instance in front of the method call and automatically pass self

```
my_inst = MyClass()
my_inst.method(arg1,arg2)
```

- `my_inst` is an instance, so the method is bound

unbound methods

- it is also possible to call a method without Python binding `self`. In that case, the user has to do it.
- unbound methods are called as part of the class but `self` passed by the user

```
my_inst = MyClass()
MyClass.method(my_inst, arg2, arg3)
```

- `self` is passed **explicitly** (`my_inst` here)!

Why???

- Consider an example. We want to specialize a new class as a subclass of list.

  ```
  class MyClass(list):
  ```

- easy enough, but we want to make sure that we get our new class instances initialized the way they are supposed to, by calling `__init__` of the super class

Why call the super class init?

If we don’t explicitly say so, our class may inherit stuff from the super class, but we must make sure we call it in the proper context. For example, our __init__ would be:

```
def __init__(self):
    list.__init__(self)
# do anything else special to MyClass
```

explicit calls to the super

- we explicitly call the super class constructor using an unbound method (why not a bound method????)
- then, after it completes we can do anything special for our new class
- We **specialize** the new class but inherit most of the work from the super. Very clever!

Gives us a way to organize code

- specialization. A subclass can inherit code from its superclass, but modify anything that is particular to that subclass
- over-ride. change a behavior to be specific to a subclass
- reuse-code. Use code from other classes (without rewriting) to get behavior in our class.

Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to improve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.
9. Make sure your class does the right thing.