Constraint Programming

Constraint programming is an example of a programming paradigm known as *declarative programming*. Declarative programming is generally characterized by the fact that rather than describing a *solution* to the computer, we describe the *problem* and "ask" the computer for the solution.

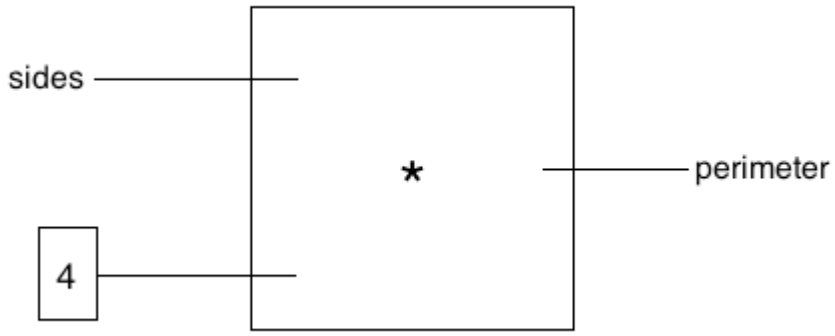
Of course, nothing happens by magic. The first step of creating a declarative programming system involves engineering a general solver for a certain type of problems, and creating a language or API (application programming interface) to programatically define a problem. After this initial effort, however, we can treat the solver as a black box, where a user describes the problem and it gives us the answer. There are many examples of real-life applications of declarative programming, such as linear programming, constraint satisfaction problems, machine learning, etc.

Now let's take a look at our constraint programming system. It has two interacting components:

- Connectors - things that store values
- Constraints - things that specify relationships between connectors

Think of the connectors as the "variables" and the constraints as the equation(s) describing the relationship between these variables. To keep the system coherent, each connector knows all the constraints it is connected to. Whenever its value changes, it notifies all relevant constraints, which propagate the message onto other connectors.

Examples:
Let's look at a very simple example, and trace through what our system does. Consider the problem where we need to find the perimeter of a square given its side length, and vice versa. The equation for this is 4 * s = p, which we represent as



So we have 3 connectors, sides, perimeter, and a constant 4. We can write this in code as

```
>>> sides = make_connector("sides")
>>> perimeter = make_connector("perimeter")
>>> def make_converter(s, p):
...     c = make_connector()
...     constant(c, 4)
...     multiplier(s, c, p)
...
>>> make_converter(sides, perimeter)
```

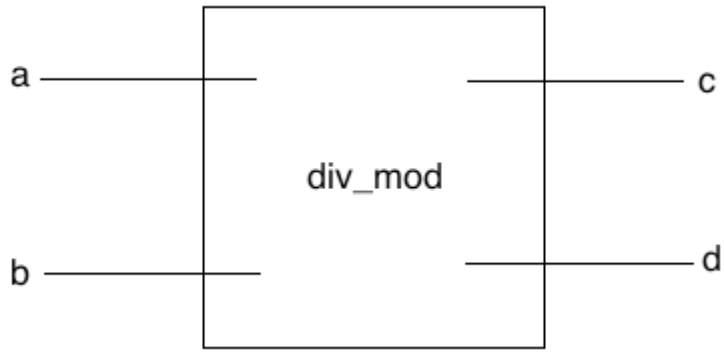
We can now figure out the perimeter of a square given the side length, and visa versa.

```
>>> sides['set_val']('user', 5)
side = 5
perimeter = 20
>>> sides['forget']('user')
side is forgotten
perimeter is forgotten
>>> perimeter['set_val']('user', 20)
perimeter = 20
sides = 5.0
```

QUESTIONS

1.) Use constraint programming to write a linear equation solver for `y = a*x+b`.

2.) Write the constraint `div_mod`, which takes two input connectors `a` and `b`, and two output connectors `c = a // b` and `d = a % b`. (Space is provided on the next page).



3.) Use the div_mod constraint to re-write the linear equation solver from question 1.

Object Oriented Programming

Last week you were introduced to the programming paradigm known as Object Oriented Programming. If you've programmed in a language like Java or C++, this concept should already be familiar to you.

Object oriented programming is heavily based on the idea of data abstraction. In object oriented programming we have:

- Classes -- Blueprints for creating types of objects. These can be though of as the abstract data types in an object oriented program.
- Objects -- These are the actual pieces of data in the program. You sometimes hear of objects being called **instances**, which means that it is an object belonging to a certain class (we say things like "object X is an instance of the Y class").

Objects can be thought of as "smart data." Objects carry around **methods** which are functions that you can call to have the object perform an action. In addition to methods, an object has access to two types of values:

- Instance variables -- these are values that the object knows and can change.
- Class variables -- these are values that all objects of a class (data type) know and share. If any object of the class changes the class variable, all other objects in the class will see the same new value.

When defining a class, we use the following syntax:

```
class OurClass(ParentClass):
    """Definition of class here (methods and class variables)."""
```

Where `OurClass` is the name of the new class and `ParentClass` is the name of the class it inherits from (we'll talk more about inheritance in lecture this week).

To define a method, we write it almost exactly the same way as when we define functions but we always include one argument before all other arguments, `self`, which is used to refer to the instance we used to call the method.

Finally, to use a class or instance's **attributes** (methods and variables), we use "dot notation." The way dot notation works is that we refer to the method or variable, `bar`, of a class or instance, `foo`, by saying: "`foo.bar`" which says "I want `foo`'s attribute `bar`."

Example:

As a starting example, consider the classes `Skittle` and `Bag`, which is used to represent a single piece of Skittles candy and a bag of skittles respectively.

```
class Skittle(object):
    """A Skittle object has a color to describe it."""
    def __init__(self, color):
        self.color = color
```

```
class Bag(object):
    """A Bag is a collection of skittles. All bags share the number
    of Bags ever made (sold) and each bag keeps track of its skittles
    in a list.
    """
    number_sold = 0

    def __init__(self):
        self.skittles = []
        Bag.number_sold += 1

    def tag_line(self):
        """Print the Skittles tag line."""
        print("Taste the rainbow!")

    def print_bag(self):
        print([s.color for s in self.skittles])

    def take_skittle(self):
        """Take the first skittle in the bag (from the front of the
        skittles list).
        """
        skittle_to_eat = self.skittles[0]
        self.skittles = self.skittles[1:]
        return skittle_to_eat

    def add_skittle(self, s):
        """Add a skittle to the bag."""
        self.skittles.append(s)
```

QUESTIONS

1.) What does Python print for each of the following:

```
>>> toms_bag = Bag()
>>> toms_bag.print_bag()
```

_____

```
>>> toms_bag.add_skittle(Skittle("blue"))
>>> toms_bag.print_bag()
```

_____

```
>>> toms_bag.add_skittle(Skittle("red"))
>>> toms_bag.add_skittle(Skittle("green"))
>>> toms_bag.add_skittle(Skittle("red"))
>>> toms_bag.print_bag()
```

_____

```
>>> s = toms_bag.take_skittle()
>>> print(s.color)
```

_____

```
>>> toms_bag.number_sold

_____________________________

>>> Bag.number_sold

_____________________________

>>> akis_bag = Bag()
>>> akis_bag.print_bag()

_____________________________

>>> toms_bag.print_bag()

_____________________________

>>> Bag.number_sold

_____________________________

>>> akis_bag.number_sold

_____________________________

>>> toms_bag.number_sold

_____________________________
```

2.) What type of variable is skittles? What type of variable is number_sold?

3.) Write a new method for the Bag class called `take_color`, which takes a color and removes (and returns) a skittle of that color from the bag. If there is no skittle of that color, then it returns None.

4.) We now want to write three different classes Postman, Client, and Email to simulate email. Fill in the definitions below to finish the implementation.

```
class Postman(object):
    """Each Postman has an instance variable clients, which is a
    dictionary that associates client names with client objects.
    """
    def __init__(self):
       self.clients = {}

    def send(self, email):
        """Take an email and put it in the inbox of the client it is
        addressed to."""
        """Your code here."""









    def register_client(self, client, client_name):
        """Takes a client object and client_name and adds it to the
        clients instance variable.
        """
        """Your code here."""
```

```
class Email(object):
    """Every email object has 3 instance variables: the message, the
    sender (their name), and the addressee (their name).
    """
    def __init__(self, msg, sender, addressee):
        """Your code here."""
```

```
class Client(object):
    """Every Client has instance variables name (which is used for
    addressing emails to the client), mailman (which is used to send
    emails out to other clients), and inbox (a list of all emails the
    client has received).
    """
    def __init__(self, mailman, name):
        self.inbox = []
        """Your code here."""









    def compose(self, msg, recipient):
        """Send an email with the given message msg to the given
        recipient."""
        """Your code here."""









    def receieve(self, email):
        """Take an email and add it to the inbox of this client."""
        """Your code here."""
```