Discussion 3:
Sequences and Trees

Tammy Nguyen (tammynguyen@berkeley.edu)
February 15, 2018

Assignments

Homeworks

HW 4 due tonight at 11:59 PM

Projects

Hog composition out

Maps due next Thursday, Feb. 22

Agenda

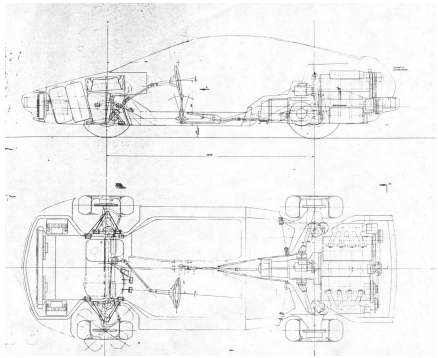
I. Data abstraction review
II. Sequences
III. Trees

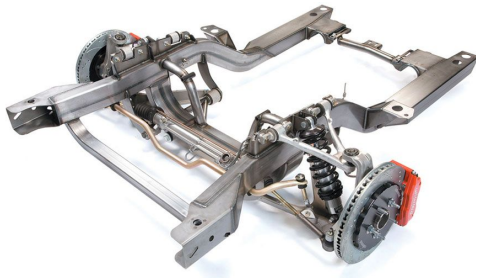
Data Abstraction

Car abstraction

abstraction barrier

What the car manufacturer sees:





What the end-user sees/uses:

ADTs

Abstract data types allow us to create and use objects (data types) with some data values instead of working with the data directly.

What makes it an ***abstract*** data type?

> They are not *real* data type in Python (e.g. numbers, functions, lists).

We implement ADTs using any data type we want, such as lists or dictionaries (or even other ADTs)!

The user sees only one side of the abstraction barrier and does not know the underlying implementation.

City abstraction

What the programmer implementing the ADT sees:

City representation:
`[name, lon, lat]`

```
def make_city(name, lon, lat):
    return [name, lon, lat]

def get_name(city):
    return city[0]

def repr_city(c):
  coords = "[" + c[1] + \
           ", " + c[2] + "]"
  return c[0] + " " + coords
```

What the end-user sees/uses:

```
make_city(name, lon, lat)

>>> get_name(city)
"Berkeley"

get_lon(city)

get_lat(city)

>>> print(repr_city(city))
Berkeley [37.9, 122.3]
```

Sequences

Sequences

```
[1, 2, 3, 4, 5]
[True, “hi”, 0]
```

Lists

sequence of values of any data type

```
“hello world”
“abcdefghijkl”
```

Strings

sequence of characters

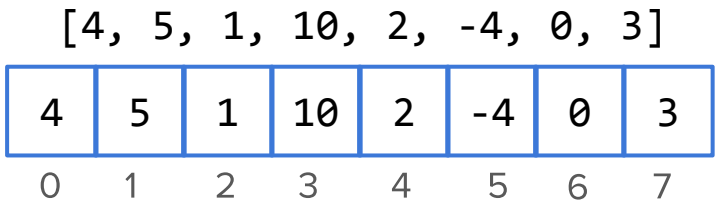
* we’ll learn other sequences, such as *tuples*, later in the course

Sequence abstraction

What we know: a sequence has a finite length, each element has a discrete position

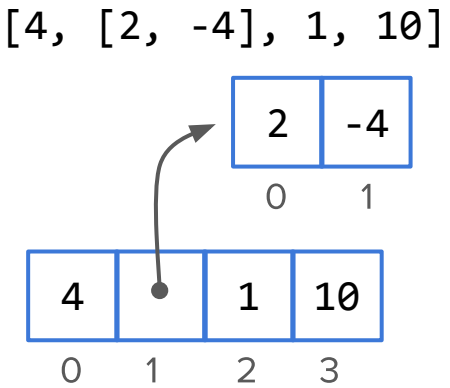
"hello" → sequence is length 5

[1, 2, 3, 4, 5] → 3rd element is 4



What we can do:

- Get the ith element
- Create a copy of a subsequence
- Check for membership
- Get the length
- Concatenate two sequences
- Iterate through the elements
- ...



What can you do with sequences?

Get item: get the ***i*th** value `<seq>[i]`

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[2]
3
>>> “hello”[3]
‘l’
```

Check membership: check if the value of `<expr>` is in `<seq>` `<expr> in <seq>`

```
>>> 3 in [1, 2, 3, 4, 5]
True
>>> ‘z’ in “socks”
False
>>> 2 + 4 in [7, 6, 5, 4, 3]
True
```

Copy subsequence: create a copy of the sequence from ***i*** to ***j*** `<seq>[i:j:skip]`

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[1:4]
[2, 3, 4]
>>> “lolololololol”[3::2]
‘ooooo’
```

Concatenate: combine two sequences into a single sequence `<seq1> + <seq2>`

```
>>> [1, 2, 3] + [4, 5]
[1, 2, 3, 4, 5]
>>> “hello “  + “world”
“hello world”
```

Iterating through sequences

You can use a **for statement** to iterate through the elements of a sequence:

```
for <name> in <seq>:
    <body>
```

For each element in <seq>:

1) Bind it to <name>
2) Execute <body>

```
i = 0
for elem in [8, 9, 10]:
    print(i, ":", elem)
    i += 1
```

Output

```
0 : 8
1 : 9
2 : 10
```

List comprehensions

You can create a list out of a sequence using a **list comprehension:**

```
[<expr> for <name> in <seq> if <cond>]
```

1) Create an empty list
2) For each element in <seq>:
    [a] Bind it to <name>
    [b] Evaluate <cond>
    [c] If <cond> evaluates to a True value, evaluate <expr> and append its value to the list
3) Return the list

```
>>> [c + "0" for c in "cs61a"]
['c0', 's0', '60', '10', 'a0']

>>> [e for e in "skate" if x > "l"]
['s', 't']

>>> [x ** 2 for x in [1, 2, 3]]
[1, 4, 9]
```

Try it out!

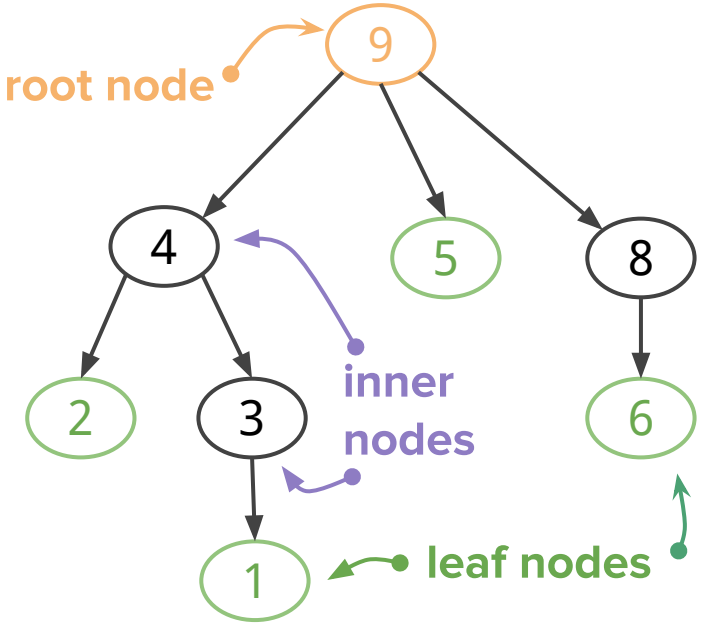
Problem 1.2 - List slicing

Problem 2.1 - List comprehensions

Trees

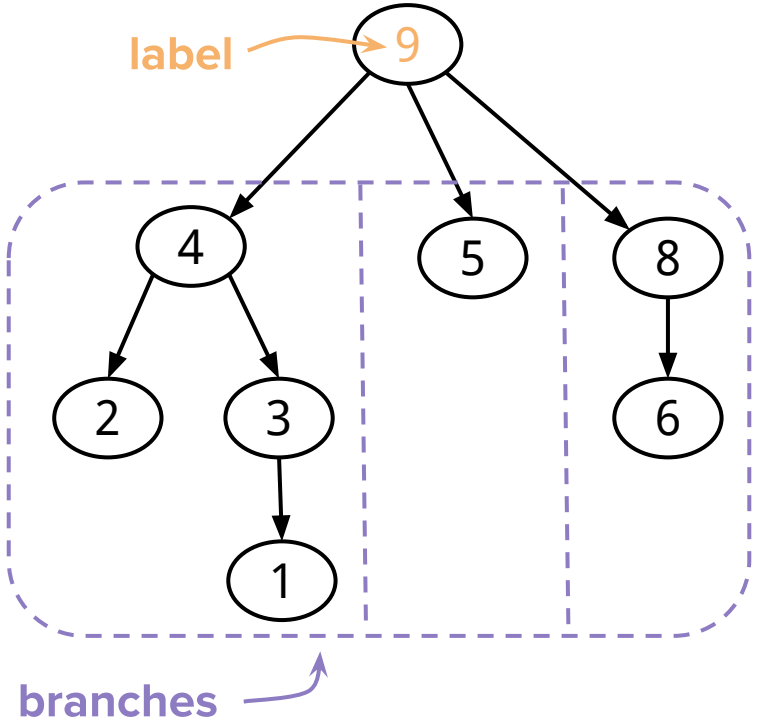
Trees abstraction: nodes and edges

- A **tree** is an abstract data type that represents hierarchical data.
- A tree is made up of nodes connected by **edges**
- A tree has three types of nodes:
  - The **root node** is at the top of the tree
  - **Leaf nodes** are at the bottom of the tree
  - **Inner nodes** are neither a root or a leaf



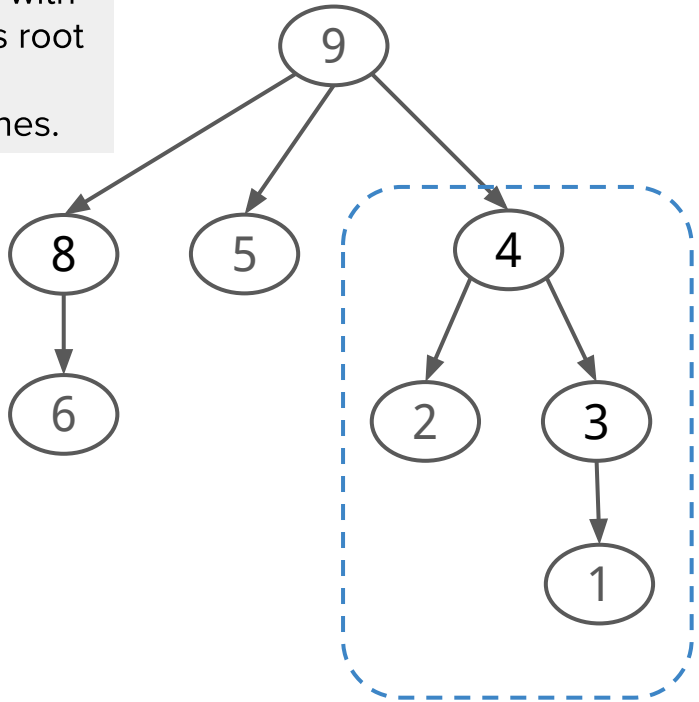
Trees abstraction: label and branches

- A tree can be defined as having a **root** and **branches**.
- The value at a tree's root is known as the **label**.
- Each edge coming out of the root points to one **branch** of the tree.
- *Recursive definition:* each **branch** of the tree is also a tree!
- Collectively, the **branches** of a tree are all the trees that are coming out of the root.

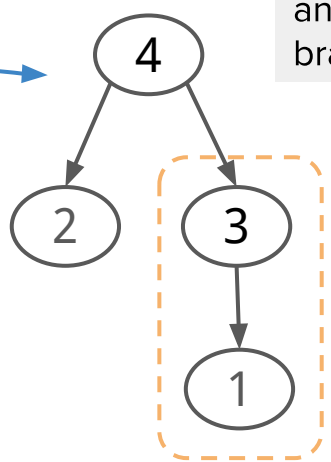


Trees are recursive!

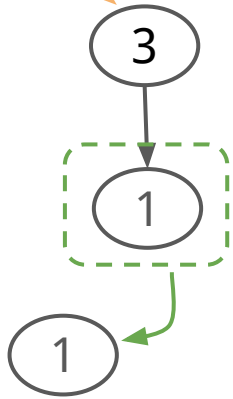
A tree with 9 at its root and 3 branches.



A tree with 4 at its root and 2 branches.

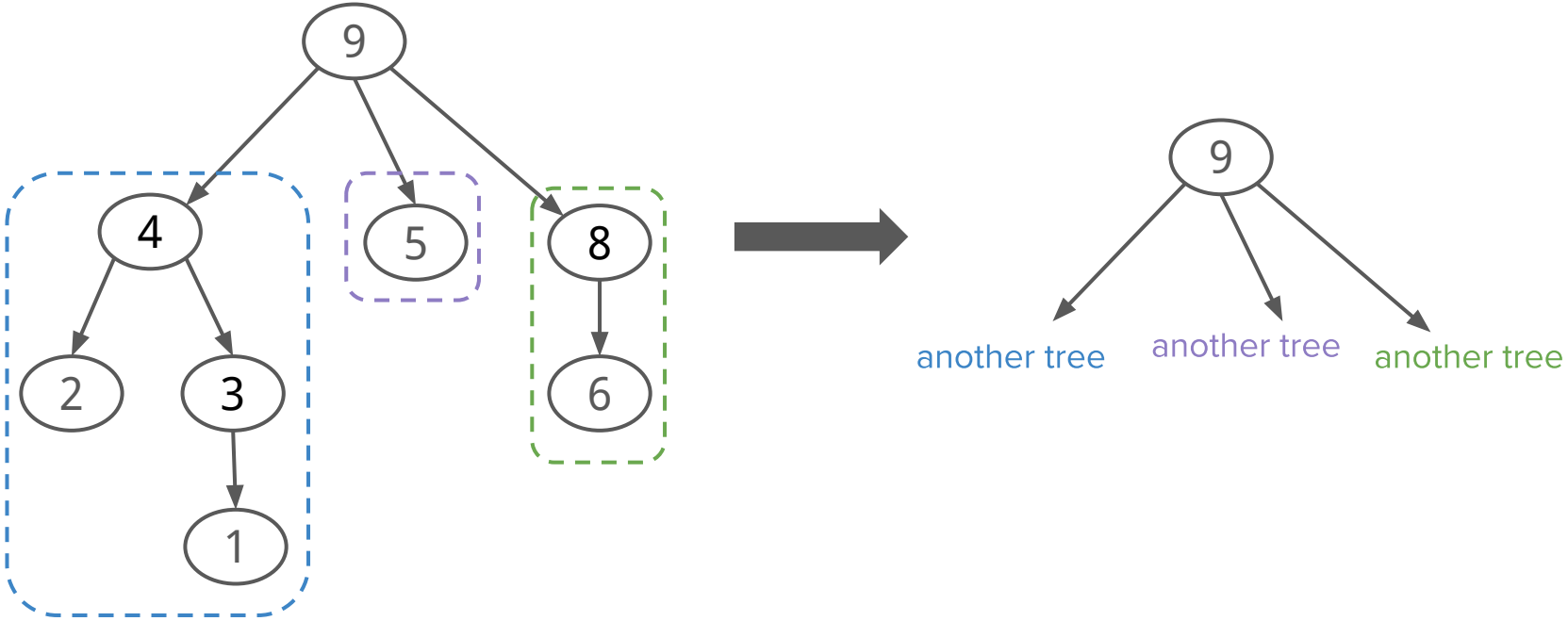


A tree with 3 at its root and 1 branch.



A tree with 1 at its root and no branches (a leaf!)

Trees are recursive!



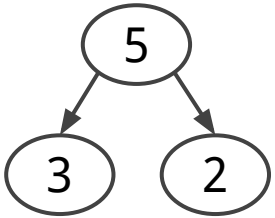
Tree ADT: Constructor

`tree(label, branches=[])`

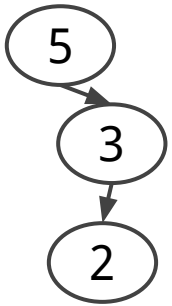
- `label`: the data value at the root of the tree
- `branches`: a list of trees immediately under the root, defaults to an empty list

We define a tree *recursively*. We specify the **value at the root** and a **list of the branches of the root** (which have their own labels and branches).

```
>>> tree(5, [tree(3), tree(2)])
```




```
>>> tree(5, [tree(3, [tree(2)])])
```

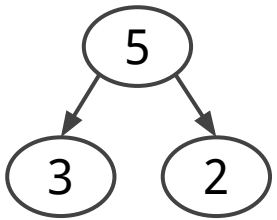


Tree ADT: Selectors

`label(tree)`: Returns the value at the root of the tree

`branches(tree)`: Returns a ***list*** of the tree's branches

```
>>> t = tree(5, [tree(3), tree(2)])
```

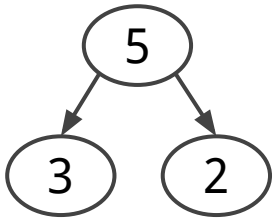



```
>>> label(t)
5
>>> type(branches(t)) == list
True
>>> [label(t) for b in branches(t)]
[3, 2]
```

Tree ADT: Convenience function

`is_leaf(tree)`: Returns `True` if the tree is a leaf (has no branches) and `False` otherwise

```
>>> t = tree(5, [tree(3), tree(2)])
```




```
>>> is_leaf(t)
False
>>> is_leaf(branches(t)) #Invalid
False
>>> is_leaf(branches(t)[0])
True
```

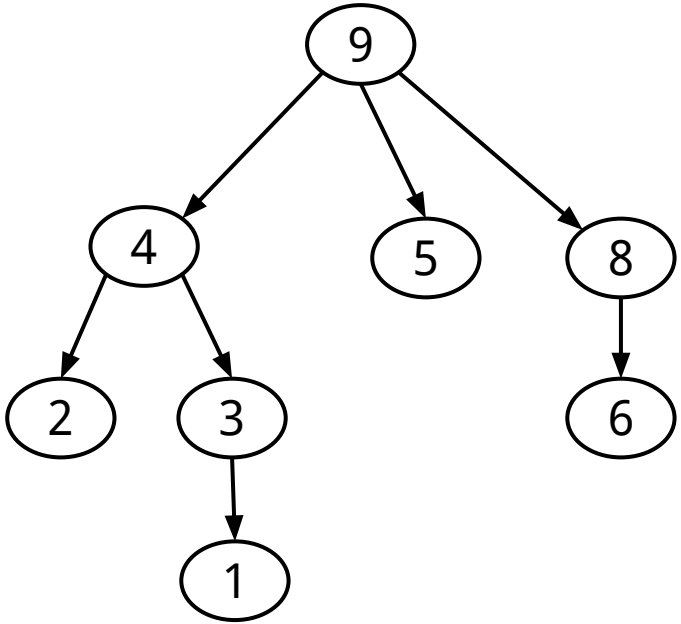
Tree abstraction

What we know:

- a tree has a root and a list of branches
- each branch is a tree object!
  - recurse...

What we can do:

- Get the value at the root
- Get the list of branches
- Iterate through the list of branches
- ... recursion (on the branches)???




```
def print_tree(tree):
    print(label(tree))
    for b in branches(tree):
        print_tree(b)
```

Processing Trees

We can take advantage of the recursive structure of trees when processing them.

Example: `square_tree(t)`

Input: a tree ADT object
Output: another tree object with the same nodes as the input tree but with each value squared

We're going to have to use the tree constructor to create the tree that we're going to return.

```
tree(label, branches)
```

Square tree

Squaring the label is easy:

```
label(t) * label(t)
```

How do we take care of the rest of the tree, a ***list*** of branches?

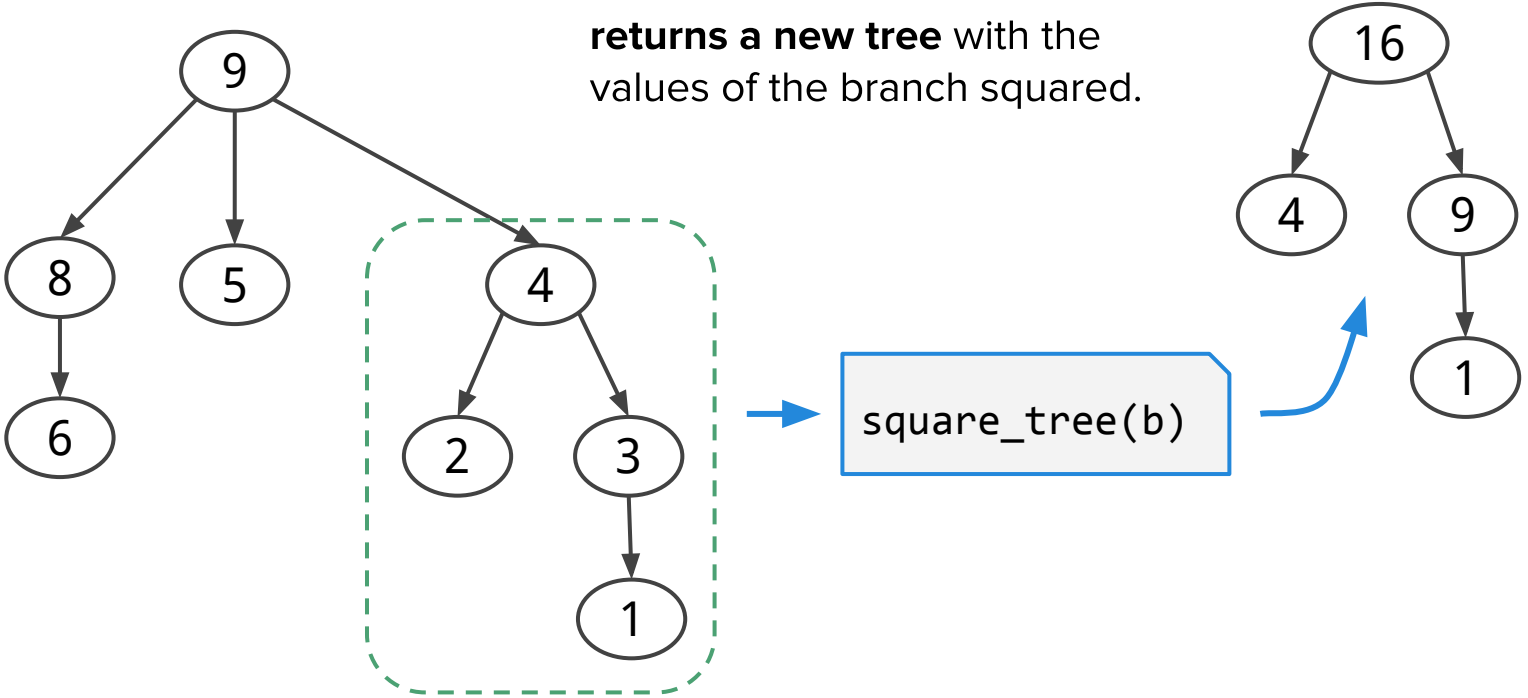
We know that each branch is itself a tree object.

We know that our function will take in a tree and returned a squared tree.
Do we know how our function will do this? Nope, and we don't care.

The **leap of faith** allows us to call `square_tree` within its body and assume it works!

Leap of faith

A recursive call on a branch **returns a new tree** with the values of the branch squared.



Appendix

Data Abstraction

Abstraction

Abstractions hide complex details.

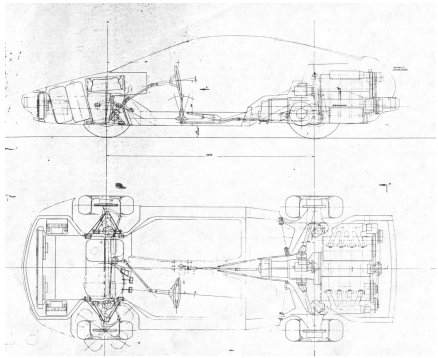
When someone asks you where you live, you don’t say (37.8716° N, 122.2727° W), you say the name of your city or general location.

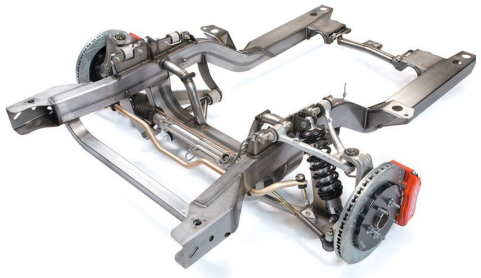
When you drive your car, you do not need to manually spark your engine or turn your wheels, you just put your key into the ignition and step on the gas pedal.

Car abstraction

abstraction barrier

What the car manufacturer sees:





What the end-user sees/uses:

ADTs

Abstract data types allow us to create and use objects (data types) with some data values instead of working with the data directly.

What makes it an ***abstract*** data type?

> They are not *real* data type in Python (e.g. numbers, functions, lists).

We implement ADTs using any data type we want, such as lists or dictionaries (or even other ADTs)!

The user sees only one side of the abstraction barrier and does not know the underlying implementation.

Two ways to implement a City ADT

```
def make_city(name, lat, lon):
    """ Represents a city
        as a list. """
    return [name, lat, lon]

def get_name(city):
    return city[0]

def get_lat(city):
    return city[0]

def get_lon(city):
    return city[0]
```

```
def make_city(name, lat, lon):
    """ Represents a city
        as a dictionary. """
    return { 'name': name,
             'lat': lat,
             'lon': lon }

def get_name(city):
    return city['name']

def get_lat(city):
    return city['lat']

def get_lon(city):
    return city['lon']
```

City abstraction

What the programmer implementing the ADT sees:

City representation:
`[name, lon, lat]`

```
def make_city(name, lon, lat):
    return [name, lon, lat]

def get_name(city):
    return city[0]

def repr_city(c):
  coords = "[" + c[1] + \
           ", " + c[2] + "]"
  return c[0] + " " + coords
```

What the end-user sees/uses:

```
make_city(name, lon, lat)

>>> get_name(city)
"Berkeley"

get_lon(city)

get_lat(city)

>>> print(repr_city(city))
Berkeley [37.9, 122.3]
```