SMT-Based Binary Analysis Implementation Manual

Möbius Strip Reverse Engineering

April 19, 2018

Contents

This document, accompanying the SMT-based binary program analysis training course, describes the design and implementation of a fully-functional binary analysis framework written in Python. Along the way, we shall also introduce programming concepts used while writing programs that analyze other programs.

1 Python Basics

1.1 Basic Data Structures

1.1.1 Lists

Lists are one of the most basic types on Python. They support the `len` method to return their length.

```
>>> x = [1,2]
>>> y = [3,6,9,12]
>>> z = ["String",None,True]
>>> len(y)
4
>>>
```

You can compare lists for equality, element-by-element:

```
>>> x == y
False
>>> x == [1,2]
True
>>>
```

To insert elements into a list, the API provides two interfaces: a **mutable** interface (whose functions modify an existing list in place), and an **immutable** one (whose functions create new lists).

- Mutable:
  - Appending elements: `l.append(e)`
  - Concatenating lists: `l1.extend(l2)`
- Immutable:
  - Appending elements: `l + [e]`
  - Concatenating lists: `l1 + l2`

```
>>> x+y
[1, 2, 3, 6, 9, 12]
>>> x.append(3)
>>> x
[1, 2, 3]
>>> y.extend([15,18])
>>> y
[3, 6, 9, 12, 15, 18]
>>>
```

They are **indexable**, including from the end if a negative number is supplied.

```
>>> x[0]
1
>>> x[-1],y[-2]
(2, 9)
>>>
```

You can **unpack** lists by providing as many variables as there are elements in the list.

```
>>> l = [1,2,3]
>>> c1,c2,c3 = l
>>> c1
1
>>>
```

You can take **slices** of lists. For positive integers `a` and `b`, use an expression like $\cdots$ to retrieve the elements from $\cdots$:

- `y[a:b]`: position `a` up to (but not including) position `b`. If `a >= b`, the result is `[]`.
- `y[:b]`: the beginning up to (but not including) position `b`.
- `y[a:]`: position `a` up to the end of the list.
- `y[a:-b]`: position `a` up to the end of the list, not including the last `b` elements.

```
>>> z = y[1:3]
>>> z
[6, 9]
```

You can create a list of a specified length, whose contents are identical, using the `*` (multiplication) syntax:

```
>>> z = [None]*5
>>> z
[None, None, None, None, None]
```

1.1.2 Tuples

Tuples are two or more objects collected into a single object.

```
>>> x = (1,2)
>>> y = (3,6,9,12)
>>> z = ("String",None,True)
>>> len(z)
3
>>>
```

They are **indexable**, including from the end if a negative number is supplied.

```
>>> x[1]+y[0]
5
>>>
```

You can create them from lists.

```
>>> tuple([1,2,3,4])
(1, 2, 3, 4)
>>>
```

You can **unpack** tuples by providing as many variables as there are elements in the tuple.

```
>>> t = (1,2,3)
>>> c1,c2,c3 = t
>>> c1
1
>>>
```

If you have a function that takes as many arguments as there are elements in a tuple, you can "unpack" the tuple conveniently with the * (**star**) syntax while calling the function. (Technically, the star syntax also works for lists.)

```
>>> def f(a,b,c): return a+b+c
>>> args = (1,2,3)
>>> f(args)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 3 arguments (1 given)
>>> f(*args)
6
>>>
```

1.1.3 Sets

Sets are collections of items, where only one of a given item can be present in the set. In other words, duplicates are removed automatically upon creating a new set or adding an element to an existing one.

```
>>> empty = set()
>>> empty
set([])
>>> x = {1,2}
>>> y = {1,2,2}
>>> z = {1,2,2,3}
>>> x == y, y == z
(True, False)
```

Sets support the following operations:

- **Intersection**: find the elements that are common to two sets.
- **Union**: combine the elements from two sets.

As with lists in section 1.1.1, Python provides mutable and immutable APIs for manipulating sets.

- Mutable API:
  - `s1.update(s2)` (alternatively `s1 |= s2`).
  - `s1.intersection_update(s2)` (alternatively `s1 &= s2`).
  - Adding an element `x` to a set: `s1.add(x)`.
  - Removing an element `x` from a set: `s1.remove(x)`.
- Immutable API:
  - `s3 = union(s1,s2)` (alternatively `s3 = s1 | s2`).
  - `s3 = intersection(s1,s2)` (alternatively `s3 = s1 & s2`).

1.1.4 Dictionaries

Dictionaries allow objects (called **values**) to be associated with other objects (called **keys**).

```
>>> empty = dict()
>>> empty
{}
>>> empty["energy"] = True
>>> empty["will"]   = False
>>> empty[0]        = "Yes"
>>> print empty[0]
Yes
>>> print empty
{0: 'Yes', 'energy': True, 'will': False}
>>>
```

Observations from the above:

- To create an empty dictionary, use `{}` or `dict()`.
- Dictionaries support adding mappings like `empty[0] = "Yes"`.
- Dictionaries support retrieving mappings like `print empty[0]`.
- The key and value objects need not have the same types as other elements in the dictionary. For example, the first two insertions into `empty` used strings for keys, and booleans for values, while the third insertion used an integer for a key, and a string for a value.

```
>>> d = {1:2,3:4}
>>> print 1 in d
True
>>> print d[1]
2
>>> d[1] = 3
>>> print d[1]
3
>>> d.update([(1,4),(2,0)])
>>> print d
{1: 4, 2: 0, 3: 4}
>>> del d[1]
>>> print d
{2: 0, 3: 4}
>>>
```

Observations from the above:

- To create a dictionary with pre-specified contents, use an expression like `{1:2,3:4}`.
- You can query whether a key `k` is present in a dictionary `d` with `k in d`.
- You can over-write the existing value associated with a key with the same syntax as adding a mapping between a key and a value.
- Given a list `l` of `(key,value)` tuples, you can use the **`dict`** method `d.update(l)` to add all of them to the dictionary `d` simultaneously.
- To remove the mapping associated with key `k`, use **`del`** `d[k]`.

```
>>> print d[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 4
>>> if 4 in d: print "Found"
... else: print "Not found"
Not found
>>>
```

Python will raise a `KeyError` exception if you try to retrieve an element that does not exist within a dictionary. (Exceptions are discussed in more depth in section 1.7.)

1.1.5 Iterating over Containers

In Python, containers that support the iterator protocol can be iterated item-by-item with the **`for`** expression. All of the data structures described previously support iteration.

```
>>> for x in (1,2,3): print x,
1 2 3
>>> for x in [4,5,6]: print x,
4 5 6
>>> for x in {7,8,9}: print x,
7 8 9
>>> for x in {1:2,3:4}: print x,
1 3
>>>
```

Note that in the case of a dictionary, only the keys will be retrieved with an ordinary `for` expression. If you want the keys and the values, use the `dict` method `items` to obtain a list of them.

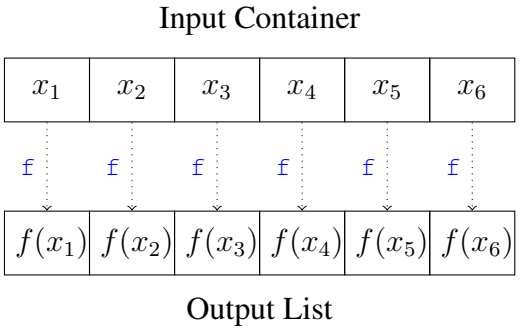
```
>>> for (k,v) in {1:2,3:4}.items():
...   print k,v
1 2
3 4
>>>
```

1.2 Transforming Containers

1.2.1 Map

The function `map(f,container)` creates a new list by applying the function `f` to each element of the container.

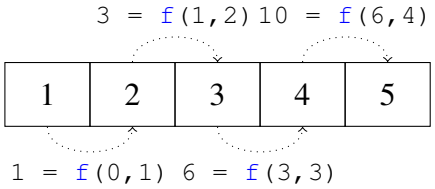
```
>>> def f(i):
...   return (i,i+1)
>>> map(f,[1,2])
[(1, 2), (2, 3)]
```



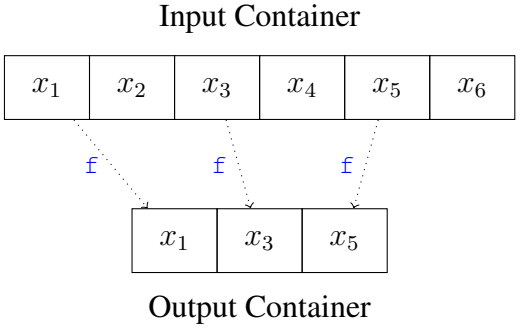
1.2.2 Filter

The function `filter(f,container)` creates a new container from those elements for which the function `f` returns `True`.

Figure 1: Behavior of `reduce`




```
>>> def f(i):
...   return i&1 == 0
>>> filter(f,(1,2,3,4,5))
(2,4)
```



1.2.3 Reduce

The function `reduce(f,container,initial)` "reduces" a new container into one element by repeatedly applying `f` to the elements. An example of its action can be seen in figure 1.

```
>>> def f(a,b):
...   return a+b
>>> reduce(f,[1,2,3,4,5],0)
15
```

- If `initial` is provided (i.e., not `None`), the sequence of actions is roughly this:
  1. `result = f(initial,container[0])`
  2. `result = f(result,container[1])`
  3. `result = f(result,container[2])`
  4. ...
  5. `result = f(result,container[len(container)-1])`
- If `initial` is not provided (i.e., is `None`), the sequence of actions is roughly this:
  1. `result = f(container[0],container[1])`
  2. `result = f(result,container[2])`

3. ···
4. `result = f(result,container[len(container)-1])`

- If `initial` is not provided and the container has one element, return that element as-is.

1.2.4 Lambda Functions

Python's **`lambda`**s allow concise definitions of small functions, as in figure 2.

Figure 2: Stand-Alone Functions vs. **`lambda`**s

```
def f(i):
  return (i,i+1)        map(lambda i: (i,i+1),[1,2])
map(f,[1,2])

def f(i):
  return i&1 == 0       filter(lambda i: i&1==0,[1,2,3])
filter(f,[1,2,3])

def f(a,b):
  return a+b            reduce(lambda a,b: a+b,[1,2],0)
reduce(f,[1,2],0)
```

Exercise: See exercise 1 in the exercise manual.

1.3 Comprehensions

Comprehensions are a very concise way to create new container objects from existing iterable objects. Python supports list, set, and dictionary comprehensions. The basic form of a list comprehension uses the syntax `[ expression for element in iterable ]`. Some examples follow; note that they could be written using **`map`**.

- `[(i,i+1) for i in [1,2]] == [(1,2),(2,3)]`
- `[i*2 for i in [1,2]] == [2,4]`

Comprehensions for all types of containers optionally allow elements from the iterable to be ignored by placing an **`if`**-condition after the **`for`** expression. The following examples could also be written using both **`map`** and **`filter`**.

- `[(i,i+1) for i in [1,2,3,4] if i&1 == 0] == [(2,3),(4,5)]`
- `[i*2 for i in [0,1,2] if i != 0] == [2,4]`

Set comprehensions are identical to list comprehensions, except they use curly brackets (`{,}`) instead of square ones (`[,]`).

- `{(i,i+1) for i in [1,2]} == {(1,2),(2,3)}`

- `{i*2 for i in [1,2]} == {2,4}`

Dictionary comprehensions require that the expression produce both keys and values, like `{ key:value for element in iterable }`. Some examples:

- `{ i:i+1 for i in [0,1] } == {0:1,1:2}`
- `{ s:len(s) for s in ["a","ab"] } == {"a":1,"ab":2}`

1.4 Iterators

In section 1.1.5, we showed how standard containers like lists, tuples, sets, and dictionaries supported iterating over the contents using syntax like `for x in int_list:`. Iterators, like the container transformation functions and comprehensions discussed previously, are very powerful tools that can simplify your programming tremendously. Iterators have been specially designed to consume as little memory as possible. They are also performance-optimized, so using iterators is likely to be faster than manually iterating over containers. Python provides the `itertools` module, allowing iterators to be combined and manipulated in variety of interesting and useful ways.

1.4.1 Combining Iterators

If we have two iterators `i1` and `i2`, a common thing to do with them might be to iterate all of the elements from `i1`, and then iterate the elements from `i2`. We can simplify this process by **chaining** `i1` and `i2` into a single iterator `iall`: `iall = itertools.chain(i1,i2)`.

```
>>> import itertools
>>> x = [1,2]
>>> y = [3,4]
>>> g = itertools.chain(x,y)
>>> for i in g:
...   print i,
1 2 3 4
```

`itertools.chain.from_iterable` works similarly, except it takes one parameter, an iterable (such as a list), where each of the elements in the iterable are themselves iterable (for example, the iterable is a list of lists).

```
>>> x = [1,2]
>>> y = [3,4]
>>> g = itertools.chain.from_iterable([x,y])
>>> for i in g:
...   print i,
1 2 3 4
```

Another operation we may wish to perform upon two iterators is to create all pairs of their possible outputs (known in mathematics as the **Cartesian product**). `itertools.product` implements this idea, as in the below.

```
>>> x = [1,2]
>>> y = [3,4]
>>> g = itertools.product(x,y)
>>> for i in g:
...   print i,
(1, 3) (1, 4) (2, 3) (2, 4)
>>>
```

In fact, you can pass more than two iterators to `itertools.product` to obtain all tuples of the iterators' possible outputs.

1.4.2 Mapping Iterators

Recalling our friend **`map`** from 1.2.1, sometimes we might want to transform the output of an iterator. `itertools` provides `imap`. `imap` assumes that the iterator returns a single value, and passes that directly to the provided mapping function.

```
>>> x = [1,2]
>>> y = [3,4]
>>> g = itertools.product(x,y)
>>>
>>> # Compare to below
>>> # Parentheses (one parameter, a tuple)
>>> g1 = itertools.imap(lambda (x,y): x+y,g)
>>> for i in g1:
...   print i,
4 5 5 6
```

`itertools` also provides `starmap`. `starmap` assumes that the iterator yields a tuple, which it then "unpacks" using the $\star$ (star) operator (recalling section 1.1.2), and then calls the provided mapping function with multiple arguments.

```
>>> g = itertools.product(x,y)
>>>
>>> # Compare from above
>>> # No parentheses (two parameters)
>>> g2 = itertools.starmap(lambda x,y: x+y,g)
>>> for i in g2:
...   print i,
4 5 5 6
```

1.4.3 Filtering Iterators

`itertools` also provides support for **`filter`**ing some elements from the iterator. `ifilter(f,i)` applies the function `f` to all elements from the iterator `i`, and keeps those for which `f` yields `True`.

```
>>> x = [1,2,3,4,5,6,7,8]
>>> g = itertools.ifilter(lambda x:x&1==0,x)
>>> for i in g:
...   print i,
...
2 4 6 8
```

`ifilterfalse(f,i)` works the same way, except it keeps those for which `f` returns `False`.

```
>>> g = itertools.ifilterfalse(lambda x:x&1==0,x)
>>> for i in g:
...   print i,
...
1 3 5 7
```

Exercise: See exercise 2 in the exercise manual.

1.5 Python Classes

Before proceeding, note that all of the code snippets shown in this section can be found in `Classes.py`. You can follow along if you like.

As in most programming languages, an **class** is a collection of data items and functions (called **class methods**) that constitute a single unit. A "class" is the abstract description of a type, whereas an **object** is an actual instance of that data type stored in memory.

Declaring classes and creating object instances is simple enough. The syntax for a class declaration is `class Name(object):`, followed by an indented sequence of function declarations and perhaps data declarations. (Although we will shortly talk about circumstances where `object` will be replaced within the class declaration `class Name(object)` – these situations are called inheritance – to use Python 2.7's "new-style" classes, always use `class Name(object)`, and never something like `class Name:` or `class Name():`, as this will trigger "old-style" behavior and also a plague of locusts.)

Below is a simple example of declaring a class `Basic`, creating an instance of it called `b`, and invoking `Basic`'s method `DoSomething` upon `b`.

```
>>> class Basic(object):
...   def DoSomething(self,i):
...     return 10+i
...
>>> b = Basic()
>>> b.DoSomething(5)
15
>>>
```

One thing to notice in the snippet above is that the class method declaration for `DoSomething` takes an object called `self` as its first parameter and an integer `i` as its second parameter. However, when we invoke the method using `b.DoSomething(5)`,

we only supply the integer parameter `i`. Behind the scenes, the line `b.DoSomething(5)` passes `b` as the `self` parameter before the others. Hence, when `Basic.DoSomething` executes, `self` references `b`.

Classes can store data inside of them. To access the data member `data`, class methods must reference `self` in doing so, as in `self.data`. Code outside of the class can also refer data members within an object, as in `h.data` in the last line of the snippet below.

```
>>> class HasData(object):
...   def Store(self,v):
...     self.data = v
...   def Retrieve(self):
...     return self.data
...
>>> h = HasData()
>>> h.Store(5)
>>> h.Retrieve()
5
>>> h.data
5
>>>
```

Note that languages like C++ allow the programmer to control which code can access class data members. In C++, data members declared `public` can be accessed freely by code outside of the class, those declared `private` can only be accessed by members of the class itself, and those declared `protected` can only be accessed by members of the class itself or derived classes. Python's support for restricting access to class data members is virtually non-existent. All class data members in Python are the equivalent of C++'s `public` members.

As an aside, although the first argument of a class method is typically called `self`, this is by convention only: that parameter can be renamed to anything, such as the shorter name `s`. The following example is equivalent to the declaration of `HasData` in the snippet above, with `self` renamed to `s`.

```
class HasData(object):
  def Store(s,v):
    s.data = v
  def Retrieve(s):
    return s.data
```

1.5.1 Inheritance

In the previous discussion on how to declare classes, we used declarations like `class HasData(object):`. Technically, this means that `HasData` is **derived** from another class called `object`. Alternatively, we say that `HasData` **inherits** the class methods from `object`. We illustrate classes that derive from something other than `object` in the example below with the classes `Base` and `Derived`.

```
>>> class Base(object):
...   def Interface(self):
...     self.InternalFunction()
...   def InternalFunction(self):
...     print "Base"
...
>>> b = Base()
>>> b.Interface()
Base
>>>
>>> # Derives from Base, above
>>> class Derived(Base):
...   def InternalFunction(self):
...     print "Derived"
...
>>> d = Derived()
>>> d.Interface()
Derived
>>>
```

Note the declaration of "Derived" above: `class Derived(Base)`. This means that the class `Derived` will automatically contain all of the functions defined upon the `Base` class. This is why we can call the `Interface` method on `Derived`, despite `Derived` not explicitly providing it: `Derived` contains that method by virtue of inheriting from `Base` (which does contain that method).

When a derived class provides different implementations for methods defined in base classes, it is said to **override** those methods. In Python 2.7 new-style classes, invoking class methods will always invoke the overridden versions. This point merits careful investigation. We observed that `Derived` doesn't explicitly implement the method `Interface`; rather, it inherits it from `Base`. When we invoke `d.Interface()`, the method `Base.Interface` invokes `self.InternalFunction()`. Which version of `InternalFunction` is invoked – `Base`'s, or `Derived`'s? The output gives the answer: `Derived`'s. The rule is that, if a derived class overrides methods that are invoked by base methods, the overridden methods will be executed rather than the ones in the base class. If you are familiar with C++, this point can be understood by saying that, in Python, all class methods are considered `virtual`.

Derived classes can still call methods from base classes, using a slightly unusual syntax: `ClassName.MethodName(self,args)`. There is an example in the snippet below: `Base.InternalFunction(self)`. Note that the class instance `self` is supplied as an argument, something that is not done in normal circumstances.

```
>>> class Derived2(Derived):
...   def OriginalInternal(self):
...     Base.InternalFunction(self)
...
>>> d2 = Derived2()
>>> d2.OriginalInternal()
Base
>>>
```

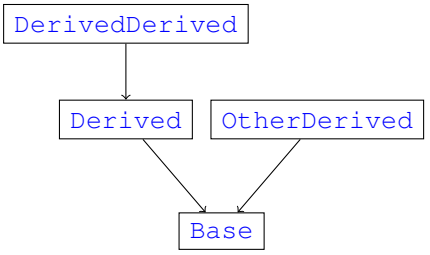
Also note in the snippet above that `Derived2` is derived from `Derived` rather than `Base`. Since `Derived` derives from `Base`, `Derived2` inherits from both.

1.5.2 Inheritance Hierarchies

The inheritance relationships between classes can be illustrated graphically as a diagram called the **class hierarchy** (or the **inheritance hierarchy**). For the class declarations below, its inheritance hierarchy can be seen in figure 4.

```
>>> class Base(object):
...   pass
>>> class Derived(Base):
...   pass
>>> class OtherDerived(Base):
...   pass
>>> class DerivedDerived(Derived):
...   pass
>>>
```

Figure 4: Inheritance Hierarchy for Example



Along these lines, Python provides a built-in function called `isinstance(x,y)`, which returns `True` if and only if the class type of object `x` is derived, somewhere along the line, from `class` type `y`. This can be understood graphically as stating that, for `isinstance(x,y)` to be true, there must be a path from the class type of `x` to class `y` in the inheritance hierarchy. In figure 4, we can see that there is a path from every class to `Base`, but there is no path from `Derived` or `DerivedDerived` to `OtherDerived`, nor vice versa. The results of the `isinstance` function, as shown in table 1, confirm this graphical intuition.

```
>>> d = Derived()
>>> isinstance(d,Base)
True
>>> isinstance(d,OtherDerived)
False
>>>
```

`isinstance` can also take a tuple of `class` types as its second parameter, as in `isinstance(x,(Derived,OtherDerived))`. It will return `True` if `x` is an instance of any of those `class` types.

Table 1: `isinstance(x,y)` (x from rows, y from columns)

	Base	Derived	OtherDerived	DerivedDerived
Base	✓	X	X	X
Derived	✓	✓	X	X
OtherDerived	✓	✓	✓	X
DerivedDerived	✓	✓	X	✓

1.5.3 Constructors and Destructors

For the objects we have created so far, we have instantiated them using lines like `d = Derived()`. The parentheses `()` indicate that a function is being called – but which function is it?

After any internal work that Python does in allocating the memory for an object, it determines whether the class provides a **constructor** – a method called `__init__`. If it does not, no function will be called and the object will be returned in its freshly-allocated state. If present, `__init__` will be called with whatever arguments were used to invoke the class. An example follows.

```
>>> class Construct(object):
...   def __init__(self,a,b):
...     self.a = a
...     self.b = b
...
>>> c = Construct(1,2)
>>> print c.a,c.b
1 2
>>>
```

The programmer is free to declare the constructor has taking as many parameters as desired. Note that only the constructor in the class that is being instantiated will be called. If the programmer wishes to call the constructor of a base class, they should do so as previously discussed in section 1.5.1: by explicitly naming the base class in a statement such as `BaseClass.__init__(self,1)`.

Destructors are the opposite of constructors. The class method `__del__` will be called when the object is garbage collected. However, given the questionable garbage collection employed by Python, this may not happen. Fortunately, in languages with garbage collection, destructors are rarely necessary. Only one of our classes shall employ a destructor, and it will be provided. Thus, we need not speak more of destructors.

1.5.4 Deriving from `object`

Earlier, we mentioned that deriving our base classes from `object` was necessary for "new-style" classes. `object` is the basic class in Python 2.7 new-style classes for any object whatsoever that the programmer might want to create. `object` implements many methods that allow classes to integrate well with the rest of Python. Derived classes may override those methods to customize their functionality. For example, `print o` will invoke `o.__str__()`. If the object does not implement this method, the method from

`object` is used. If the class does declare a method `def __str__(self)` that returns a string describing the object, we gain the ability to print our objects using the built-in facilities.

Example	`object` method	Called when . . .
`print x`	`__str__`	String printing
`repr(x)`	`__repr__`	Representation Printing
`x == y`	`__eq__`	Equality comparison
`x != y`	`__ne__`	Inequality comparison
`dict[x]`	`__hash__`	Hashing (`dict` and `set`)
`x(1)`	`__call__`	"Calling" `x`

Having a unified interface for this functionality is very useful, but it does become tedious to implement these methods for each class.

To see what happens when we don't implement the methods in the previous table, let's revisit section 1.1.4, the introduction to dictionaries. Internally, Python uses a hash table data structure to implement a dictionary. To look up an item `k` in the dictionary `d` (i.e., `d[k]`), Python:

1. Hashes `k` via `hash(k)` (which, in turn, invokes `k.__hash__()`).
2. Retrieves the list of keys `keylist` in `d` with that hash value.
3. Uses equality comparisons `k == o` (which, in turn, invokes `k.__eq__(o)`) for `o` in `keylist` to determine whether `k` is present amongst them.
4. If found, returns the data associated with `k`.

In other words, in order to be useful as keys in dictionaries, objects must implement both `__hash__` and `__eq__`. Built-in types such as integers, strings, and tuples support these operations. For classes deriving from `object`, the default behavior is to use a pointer to the object as the hash value, and use comparison of pointers for equality testing. In other words, default equality and hashing behavior treats two objects as equal if and only if they correspond to the same allocation.

The example below illustrates failures of the default behavior. We probably want to consider the variables `a` and `b` to be "equal" to one another. We probably also want them to be treated as equal when used as keys in dictionaries. Under the default behavior, neither shall be true.

```
>>> # Create a custom class
>>> class A(object):
...   def __init__(s,i):
...     s.i = i
...
>>> # Create two instances
>>> a = A(1)
>>> b = A(1)
>>>
>>> # Are they equal? No.
>>> a == b
False
>>>
>>> # Are their hashes equal? No.
>>> hex(hash(a)),hex(hash(b))
('0x22373f', '0x2256d5')
>>>
>>> # hash() treats them as different objects,
>>> # hence d contains two different bindings
>>> d = dict()
>>> d[a],d[b] = "a","b"
>>>
>>> # Representation-print d
>>> d
{<__main__.A object at 0x02256D50>: 'b',
<__main__.A object at 0x022373F0>: 'a'}
>>>
>>> # String-print a
>>> print a
<__main__.A object at 0x022373F0>
>>>
```

Note also how the dictionary is printed in response to the statement `d`: `b` is displayed as `<__main__.A object at 0x02256D50>: 'b'`. This is the default behavior of `object.__repr__()`. On the last line, `print d` yields something similar. This is the default behavior of `object.__str__()`. We will want to implement `__repr__` and `__str__` methods for many of our objects.

After defining these functions to do sensible things, we obtain the results that we desire.

```
>>> # Override the necessary built-ins
>>> class A(object):
...   def __init__(s,i): s.i = i
...   def __repr__(s):   return "A(%d)" % s.i
...   def __str__(s):    return str(s.i)
...   def __hash__(s):   return hash(s.i)
...   def __ne__(s,o):   return not s==o
...   def __eq__(s,o):
...     return type(s)==type(o) and s.i == o.i
...
>>> # Create two instances
>>> a = A(1)
>>> b = A(1)
>>>
>>> # Are they equal? Yes!
>>> a == b
True
>>>
>>> # Are their hashes equal? Yes!
>>> hex(hash(a)),hex(hash(b))
('0x1', '0x1')
>>>
>>> # Since the hashes are equal, and the
>>> # objects themselves are too, there
>>> # will be only one mapping in the
>>> # dictionary.
>>> d = dict()
>>> d[a] = "a"
>>> d[b] = "b"
>>>
>>> # Representation-print d, sensibly
>>> d
{A(1): 'b'}
>>>
>>> # String-print a, sensibly
>>> print a
1
>>>
```

After the introductory material, all classes shall come with these uninteresting boilerplate methods already provided.

Overriding `__call__(self,arg1,arg2,...)` in a class allows objects derived from that class to be *called*, as though the objects were functions. Most commonly, this functionality is used to provide something that behaves like a constructor; i.e., calling an existing object will return a new object of the same type. We will use this at several points to simplify our designs.

1.5.5 Class Properties

Let's say we wanted to implement a class representing 16-bit integers. I.e., our class contains a value `i` which we would like to restrict in the range of [0,0xFFFF].

```
class Word(object):
  def __init__(s,i):
    s.i = i & 0xFFFF
```

The value is masked with `0xFFFF` upon construction, but after that, the programmer must remember to manually mask any subsequent values stored into `Word.i`. We could define a method to set `i`:

```
class Word(object):
  def __init__(s,i):
    s.set(i)
  def set(s,i):
    s.i = i & 0xFFFF
```

However, the user can still access the `i` variable directly, thereby potentially allowing invalid values.

Python presents an elegant solution for access control of data items through what are called **properties**. Properties are functions that behave exactly the same way as variables. When the user tries to get the value of the property (as in `print o.i`), a **getter** class method is invoked. When the user tries to set the value of the property (as in `o.i = 1`), a **setter** method will be invoked. The user accesses them identically to variables, where the class enforces access control behind the scenes that the user has no business knowing about or tampering with.

The code below illustrates an implementation of properties. `def i(self)` is prefixed with `@property`, which makes it the getter for the property `i`. `def i(self,i)` is prefixed with `@i.setter`, which makes it the setter for the property `i`. Internally, these methods hold the value in variable called `_i`. The setter masks the value before storing it.

```
>>> class Word(object):
...   # Property decorator.
...   @property
...   # This is the getter.  The name of the
...   # method is the name of the property.
...   def i(self):
...     return self._i
...
...   # Property decorator.
...   @i.setter
...   # This is the setter.  The name of the
...   # method is the name of the property.
...   # It performs masking.
...   def i(self,i):
...     self._i = i & 0xFFFF
...
...   # Allow objects to be called.
...   def __call__(self,i):
...     return Word(i)
...
...   # The constructor invokes the setter.
...   def __init__(self,i):
...     self.i = i
...
>>> w = Word(0x12345678)
>>> print hex(w.i)
0x5678
>>> w.i = 0x88888888
>>> print hex(w.i)
0x8888L
>>> # Invoke __call__()
>>> x = w(1)
>>> print x.i
1
>>>
```

1.6 Simulating Enumerations

In many languages such as C, the language provides support for declaring integers that have symbolic meanings. For example, let's say we have a program that makes use of the concept of colors, and that we consider three different colors `Red`, `Blue`, and `Green`. We can declare this as an **enumeration** in C:

```
// Option #1:  Constants
const int Red = 0;
const int Blue = 1;
const int Green = 2;

// Option #2:  Enumeration
enum color_t {
  Red,
  Blue,
  Green
};
```

The advantage of using an enumeration instead of integers is that a type is created within the program, and as such, the compiler will ensure that `color_t` types are used consistently. For example, for a function with prototype `void print(color_t c)`, the compiler will signal an error if the user invokes `print(5)`, as `5` is not a value of type `color_t`. Additionally, if using integers, the programmer must remember to specify the `const` keyword, otherwise a reference to `Red` will refer to a memory location that may change and not a fixed integer as we desire.

When writing program analysis tools, we will make frequent use of enumerations. Unfortunately, by default, Python 2.7 does not support them. Python versions 3 and above do support them, and they are available in Python version 2 through packages that have been backported. To reduce the number of external dependencies required by the code in this course, we shall use a custom solution for enumerations.

We want our enumerations to have the following properties:

- Type-safety. I.e., if a function expects a certain type of value, it should be possible to ensure that a particular enumeration element is of that type.
- String-printing and representation printing.
- Retrieval of an integer, given an enumeration element.
- Creation of an enumeration element, given an integer.
- Equality and inequality comparison, and hashing.

Support for enumerations is implemented in `Enumerate.py`. The base class `EnumElt`, used to represent elements of any enumeration, derives from `object` and overrides the standard methods required for printing, comparison, hashing, and calling. The method `IntValue()` returns an integer for a given enumeration element, starting from `0` and up to the number of elements passed while creating the enumeration. For some enumeration with name `EnumName`, our code creates a new enumeration element class type in the form of a class derived from `EnumElt` named `EnumNameElt`.

A series of helper functions assist us. The function `enum_lower(name,sequential)` expects a string for the name of the enumeration, and a list of strings naming the enumeration elements. It returns a Python `type` object for the enumeration element type, and a list of enumeration elements corresponding to those values passed as strings. As indicated by the word `lower` in the function's name, the names of the enumeration elements (as returned by `__str__()`) will be in lower-case.

```
>>> ColorElt, Colors = enum_lower("Color", ["Red", "Blue", "Green"])
>>> # Unpack the list into its three elements
>>> Red, Blue, Green = Colors
>>> print str(Red), repr(Red)
red Red
>>>
```

`enum_upper` uses upper-case names for the element strings. `enum_specialstr` allows the user to provide custom strings for each enumeration element:

```
>>> EnumElts = [("One","AAAAH"),("Five","OOOO")]
>>> SpecialElt, Special = enum_specialstr("Special",EnumElts)
>>> One, Five = Special
>>> print str(One), repr(One)
AAAAH One
>>>
```

1.7 Exceptions

Exceptions are a common construct in most programming languages, used to signify that something has gone wrong while the program was executing. Python uses the terminology **raising** to describe the act of signalling an exception. An example Python's interpreter signalling an exception follows.

```
>>> def lookup(l,i): return l[i]
>>> lookup([1,2],3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in lookup
IndexError: list index out of range
>>>
```

Python implements exceptions via classes. The basic type of exception is a class called `Exception`, from which other types of exceptions derive. The standard library provides an entire class hierarchy of exceptions for ordinary circumstances, a small selection of which is shown in table 2. To raise an exception, use the **`raise`** statement followed by the exception constructor, as in **`raise`** `TypeError`.

Table 2: Python Standard Exceptions

Exception	Type of Error Signalled
`TypeError`	An incorrect type was used.
`ValueError`	An incorrect value of the proper type was used.
`IndexError`	A bad index was used (e.g., beyond the end of an array).
`NotImplementedError`	Attempt to use un-implemented functionality.
`RuntimeError`	Any unspecified error.

If you are performing some operation that may raise an exception, you may wish to catch and handle the exception. Let's say you want to return the `i`th element of a list

`l`. Indexing into the list by `i` may raise the `IndexError` exception if `i` is outside of the length of the list. Place the code that may raise the exception in a **`try`** block as shown below. If an exception occurs, Python will inspect the type of the exception, look at the types of the exceptions specified (in-order) by the **`except`** blocks following the **`try`** block, and execute the code corresponding to the first exception type that matches. After executing a **`try`** block, regardless of whether an exception was thrown (i.e., whether any of the **`except`** blocks executed), it will execute the **`finally`** block if one is present.

```
>>> def lookup(l,i):
...   try:
...     res = l[i]
...   except IndexError, e:
...     res = None
...   except ValueError, e:
...     res = []
...   finally:
...     return res
>>> print lookup([1,2],3)
None
>>>
```

We can write the same code without using a **`finally`** block:

```
>>> def lookup(l,i):
...   res = None
...   try:
...     res = l[i]
...   except IndexError, e:
...     res = None
...   except ValueError, e:
...     res = []
...   return res
>>>
```

Sometimes, you may wish to catch an exception, take some action (such as printing something to the console), and then re-raise the same exception. You can accomplish this using the **`raise`** statement with no arguments:

```
>>> def lookup(l,i):
...   try:
...     return l[i]
...   except Exception, e:
...     print "!!exn::", e
...     raise
>>> print lookup([1,2],3)
!!exn:: list index out of range
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in lookup
IndexError: list index out of range
>>>
```

Since Python uses a class called `Exception` to handle exceptions internally, programmers can derive their own classes from `Exception` to create custom exceptions. Derivatives of `Exception` should support `__str__` at a minimum, and generally do not require more than this.

```
class MyException(Exception):
  def __init__(s,string):
    s.str = string
  def __str__(s):
    return self.str
```

To raise this exception somewhere in the code, use a statement such as `raise MyException("oh no!")`.

1.8 Package Structure

For small projects, it makes sense to put all of the code files in a single directory. Once the program has grown to the point where it has separate components, it makes sense to put those components into their own directories. Once a package spans multiple directories, Python has a set of fairly rigid expectations for how the directories should be structured and what they should contain. We begin by discussing the single-directory setting, followed by the multiple-directory setting.

1.8.1 Inter-Module Dependencies in the Single-Directory Setting

If a module `A.py` needs to make use of some element `x` defined in `B.py`, this is referred to as an **inter-module dependency**, specifically a dependency of the `A` module upon the item `x` defined within the `B` module. There are several options for referring to things defined in other modules.

1. `A` imports `B` as a module. This is accomplished by placing into `A.py` an **import statement** such as `import B`. By convention, `import` statements should be located physically at the top of a `.py` file. After such a statement, a **module variable** called `B` corresponding to the contents of `B.py` is accessible from the scope of module `A`, and `A` can refer to `x` by `B.x`.

2. `A` imports `x` from `B` without importing the rest of the module. A statement such as `from B import x` is used in this case, after which `A` can just refer to `x` without referencing `B`. Note that the module `B` will not be available with this form of import, i.e., statements such as `B.y` will cause runtime errors.

3. `A` imports `x` from `B`, and renames it to something else. Since `x` is a very short and common name, it is conceivable that `A` (or some other module referred to by `A`) also defines something called `x`. In this case, there could be ambiguity to both the programmer as well as the Python run-time system to which object `x` refers to when `A` references `x`. For this case, Python provides a special `from . import . as .` syntax, as in `from B import x as B_x`. Now the object `x` from the module `B` is available within module `A` under the name `B_x`. As above, the module `B` will not be available with this form of import.

4. `A` imports everything from `B`, using a statement like `from B import *`. This option should be used with care, for the reasons discussed in the last item; that module `A` may contain items with the same names as those in `B`, and that importing everything would cause confusion as to which object of multiple objects with the same name is being referenced.

1.8.2 Invoking Modules in the Single-Directory Setting

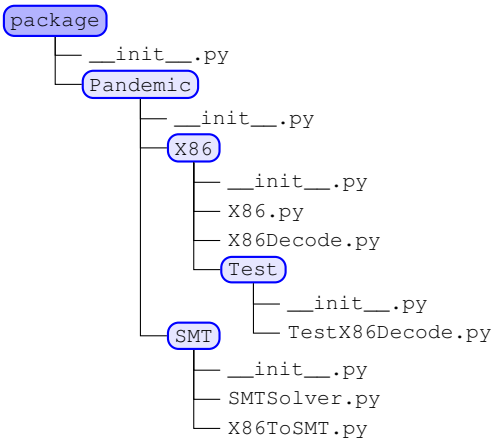
If the Python is in the user's `PATH` environment variable, that user can invoke files with a `.py` extension directly, such as by executing the command `module.py`. Alternatively, one can specify the path to the Python interpreter followed by the `.py` file that they wish to execute, as in `\python27\python.exe module.py`. When invoking a module, the Python interpreter will set a variable within the module called `__name__`. When invoking a module directly with the interpreter, as just described, this value will be set to the string `"__main__"`. Programmers often take advantage of this to implement something that behaves like `main()` in a C program. In particular, at the bottom of a `.py` file, you will often see code such as the following:

```
if __name__=="__main__":
  doSomething()
```

The module can take command-line arguments if it serves as some utility that the user may invoke directly. Another common option, used when it is not meaningful for the module to be invoked as a stand-alone program, is to run tests for the module's functionality within such an `if`-statement. (We shall discuss testing soon, in section 1.9.)

1.8.3 Multiple Directories

Figure 6: Hypothetical Directory Listing



Once our code no longer fits comfortably into a single directory, we might elect to structure it into multiple directories, as shown for example in figure 6. (Note that this directory listing is hypothetical and does not correspond to the structure of the code in this course.) Each directory starting from the package root that contains Python code (or

contains sub-directories with Python code) must contain a file called `__init__.py`, even if that file is empty (which it typically is, and all `__init__.py` files in this course will be empty).

1.8.4 Inter-Module Dependencies with Multiple Directories

The root directory of this hypothetical project is named `package`. All module names are calculated relative to this directory. For example, the file `TestX86Decode.py` can be referred to from anywhere within the codebase by the module path `Pandemic.X86.Test.TestX86Decode`. Such module paths are called **fully-qualified**, because they specify the entire path from the project root directory to the module's location. As one might imagine, it would be tedious to use these fully-qualified names everywhere within the codebase. Thus, the Python developers have included two shortcuts for common usage scenarios to eliminate the need for fully-qualified paths. All other scenarios require fully-qualified module paths.

1. Modules frequently refer to modules located in the same physical directory. For example, we might imagine that `X86Decode.py` will rely upon `X86.py`, which are both located in the same directory (namely, `X86`). It is legal, albeit verbose, to write the statement `from Pandemic.X86.X86 import *`. To reduce the tedium, the default behavior of `import X` is to first check the directory in which the module performing the `import` resides when looking for `X.py`. Hence, `X86Decode.py` can simply use `from X86 import *`, as in the single-directory scenario.

2. Another frequent case is when a module `import`s a module from the parent directory. For example, we can imagine that `TestX86Decode.py` will rely upon `X86Decode.py` in the `X86` directory, which is one level above the `Test` directory. As before, it is legal to use a fully-qualified module path in `TestX86Decode.py` such as `import Pandemic.X86.X86Decode`. As before, this is tedious and Python offers a shortcut. Prefixing a module's name with two dots in an `import` statement, as in `import ..X86Decode`, will cause `import` to check the directory above the one in which the module resides for the named module. Using more than two dots is erroneous, i.e., only the immediate parent directory can be referenced relatively like this.

1.8.5 Invoking Modules with Multiple Directories

Invoking `.py` files becomes more involved when the project contains more than one directory. As discussed in section 1.8.2, a module's `__name__` variable is set to `"__main__"` when the module is invoked directly from the command-line. Python consults a module's `__name__` variable to determine where in the package the file resides, and uses that to resolve `import` statements referencing modules in other directories. As a result, when a module in a multi-directory project is invoked directly, Python will not know where within the package the module lies, and consequently fail to resolve any reference made to modules not contained in the same directory.

The workaround is simple. Starting from the package root directory (i.e., `package` in the running example), use the `-m` option followed by a fully-qualified module name

to invoke that module. For example, from the `package` directory, a command like `\python27\python.exe -m Pandemic.X86.Test.TestX86Decoder` (note the lack of a `.py` extension in this command) will cause the code in `TestX86Encoder.py` to run.

1.9 Testing

Testing is a large and enormously important subject in software engineering. Testing provides some degree of confidence that the program does what it is supposed to do. It can reveal the presence of errors, although not guarantee their absence. Many books have been written on the topic of testing alone. We shall not re-hash them here. A piece of code that invokes another piece of code and performs a check that the result is as expected will be referred to as a **test case**. A collection of test cases shall be known as a **test suite**.

The code in this course will be heavily tested to guarantee some level of quality and correctness, as well as to provide programming exercises for the student. You are given a mostly-complete SMT-based binary program analysis framework where certain sections of the code have been replaced by statements such as `raise ExerciseError("bitblast_Add")`, and you will be expected to write those code snippets. Each exercise is accompanied by a test case; the exercise is complete when the test suite passes.

1.9.1 Testing for Type Errors

Though functional correctness testing is important for programs written in any language, it takes on a special role when the programming language has a weak type system, as in Python or Ruby: namely, to remediate deficiencies in the type system. For example, the standard library of the C programming language provides a function called `int strlen(const char *str)`, which takes as input a `char *` string and returns its length as an integer. C compilers will analyze the program's source code and ensure that any calls to `strlen` are actually provided a parameter whose type is `char *`, and that its return type is used as an integer. If these statements are ever false, the compiler will issue an error and refuse to compile the program. On the other hand, Python does not enforce types as stringently as C and other languages, so equivalently erroneous code (say, `len(1)`) shall be considered legal programs by the Python bytecode compiler. Only at run-time when this statement executes shall an exception be raised. Thus, one goal in testing Python programs is to simply execute as much of the code as possible (known in testing literature as **high code coverage**) in order to flush out run-time type errors that are invisible to the Python bytecode compiler.

1.9.2 Python's `unitttest` Module

The Python standard library contains a module named `unittest` to help programmers test their software and organize their tests. That module exports a class named `TestCase`. To use the `unittest` facilities, programmers should derive a class from `TestCase` and implement their test cases as methods defined upon the derived class.

Any method upon a class derived from `TestCase` whose name begins with the letters `test` is considered to be a test. The tests should perform their checks using the `unittest` class methods as partially shown in table 3.

Table 3: `unittest` Methods

Method	Functionality
`assertEqual(x,y,msg)`	Checks that `x` and `y` are equal. If not, the test has failed and `msg` shall be printed to the console.
`assertNotEqual(x,y,msg)`	Checks that `x` and `y` are not equal.
`assertTrue(b,msg)`	Checks that `b` is `True`.
`assertFalse(b,msg)`	Checks that `b` is `False`.
`assertIsNone(c,msg)`	Checks that `c` is `None`.
`assertIsNotNone(c,msg)`	Checks that `c` is not `None`.
`assertIsInstance(c,i,msg)`	Checks that `c` is an instance of `i`.
`assertIn(k,d,msg)`	Checks that `k` is in the collection `d`.

In this course, our test suites shall reside in modules that contain a single class derived from `TestCase`. To invoke a test suite, we follow the lead of section 1.8.5. From the root directory of the project, you will run a command such as `python -m unittest Pandemic.Solvers.Test.TestSMTSolver` to invoke the tests related to the SMT solver. The command just listed will run all of the tests in the module `Pandemic.Solvers.Test.TestSMTSolver` and report those tests that have:

- Produced an error, meaning raised an exception that was not expected by the test.
- Failed, meaning that the code ran without raising any exceptions, but did not produce the correct output.
- Passed, meaning that the code behaves properly (according to the test).

1.9.3 Programming Exercises in This Course

In this course, you will be provided with a mostly-complete framework for SMT-based binary program analysis, with portions of the code removed and replaced by statements that raise exceptions. An example is shown in the following snippet.

```
# Exercise:  just return the name of the
    variable.
def __str__(self):
  raise ExerciseError("Variable::__str__")
```

It shall be your job as a student to erase the line containing the `raise` statement and replace it with a proper solution as indicated by the surrounding comments and/or the exercise manual. You will test your solutions as in the previous section 1.9.2, by invoking the unit tests for the relevant module (and the pertinent module shall be indicated in the exercise manual). Interpret the results of the unit tests as follows:

- All exercises initially produce an error, since they are implemented by raising an `ExerciseError` exception (declared in `ExerciseError.py`). If you have removed the `ExerciseError` exception and the test still produces an error, this corresponds to an error in your code, the details of which shall be contained in the text describing the exception.
- If a test fails, this means that the code executed without raising any exceptions, but that its result was incorrect. I.e., your code contained an error.
- If the test passes, that part of the exercise is completed.

The `unittest.TestCase` class by default runs all test cases contained within the module under test, i.e., it will report the errors and failures for all components of the exercise. Since the exercises in this course often contain multiple tests (corresponding to multiple parts of the code that you must complete), this means that the test suite will produce a lot of output the first few times it is executed. To reduce the amount of output, it is advisable to use the "fail fast" feature of the `unittest` module, which causes it to stop as soon as the first error or failure is encountered. This mode is enabled by specifying the `-f` flag on the command line, as in `python -m unittest -f Pandemic.Solvers.Test.TestSMTSolver`.

1.10 Documentation

Documentation is an import part of any software project. High-quality documentation should have the following properties:

- It provides high-level insight, alleviating the need for the user to read the source code to get the gist of a particular class, function, or data item. Every module (i.e., `.py` file) should begin with a description of what purpose it serves within the project.
- It is thorough. Every class, function, and data item that is meant to be usable outside of a module is described in the documentation.
- It is cross-referenced. For example, if a function takes as argument an instance of some specific class, it is a nice touch to create a hyper-link to the declaration of that class so that the user can rapidly acquaint themselves with it.
- It contains links into the source code, so that the user can quickly peruse the implementation of something if they desire more information than what the documentation provides.
- The documentation itself is contained within the source code. This way, the documentation can be automatically generated by running a tool directly on the source code. Thus, the documentation never gets out-of-date with respect to the software, unless a lazy programmer decides not to update the documentation when they change something important.

The code within this course has documentation in-line, i.e., within the declaration of classes, class methods, functions, and data items. We use the Sphinx Documentation Generator project to automatically parse the documentation from the code and build HTML files that can be perused externally. You should keep `./docs/_build/index.html` open in your browser at all times to ease your cognitive burden as a programmer.

1.11 Foreign Functions and `ctypes`

A problem exists in the world of software development where programs written in one language might want to use libraries written in other languages. Such libraries are said to be **foreign**, given that they are written in a different language from the project wanting to use them. Interfacing with foreign functions is a very frequent use-case for Python.

Technical and scientific communities have developed highly-optimized libraries over the years, generally written in languages such as C++, FORTRAN, and/or assembly language. There are many reasons not to port these projects to Python, such as that the libraries are already mature and robust (and that porting them would throw away many years' worth of bug fixes), or the fact that pure Python is generally slower than the aforementioned languages at performing optimized algorithmic computation. On the other hand, it is easier to write code in Python than the other languages due to its permissive type system, concise syntax, extensive standard libraries, and large ecosystem of third-party libraries. If it is the case that foreign libraries consume more CPU time than the Python portion of the code, then Python's slow speed is mitigated, and using Python is a victory over the other languages. Python is very popular indeed in scientific communities for these reasons.

Relief for the problem of interfacing with C code comes from Python's standard `ctypes` library, which allows the user to:

- Load foreign C libraries into memory.
- Call functions in foreign libraries.
- Declare data types and data values that will be laid out in memory precisely the same way as they would be in C. For example, the code `(ctypes.c_uint32 * 4)(1,2,3,4)` will allocate a piece of memory corresponding to four `uint32` types (i.e., 32-bit unsigned integers) from C laid out contiguously in memory with no gaps, and where the values are `1`, `2`, `3`, and `4`.

We shall return to the `ctypes` library when our implementation requires it.

2 X86

Each instruction consists of:

1. Zero or more optional **prefixes** (dictating sizes, atomicity behavior (LOCK), repetition (REP), etc.),
2. A **mnemonic**, which gives a programmer-friendly name to the operation being performed by the processor,
3. Zero to three **operands**, the "arguments".

Accordingly, our Python API will expose a class called `Instruction` that contains:

1. A list of elements from an enumeration of prefixes;
2. An enumeration element for the mnemonic;
3. At most three objects derived from `Operand`.

2.1 Meta-Data: X86MetaData.py

`X86MetaData.py` contains several enumerations (in the style of section 1.6) describing elements within the X86 instruction set.

- Prefixes `REP` through `LOCK`
- Mnemonics `Aaa` through `Xorps`
- 8-bit registers `Al` through `Bh`
- 16-bit registers `Ax` through `Di`
- 32-bit registers `Eax` through `Edi`
- Segment registers `ES` through `GS`
- Control registers `CR0` through `CR7`
- Debug registers `DR0` through `DR7`
- FPU registers `ST0` through `ST7`
- MMX registers `MM0` through `MM7`
- XMM registers `XMM0` through `XMM7`
- EFLAGS flags `OF` through `CF`

- Memory sizes `Mb` through `Mdq` (discussed later)

This file also contains a handy piece of functionality relating to the flags: a dictionary, `FlagToEFLAGSPosition`, that specifies the proper bit within EFLAGS for each flag.

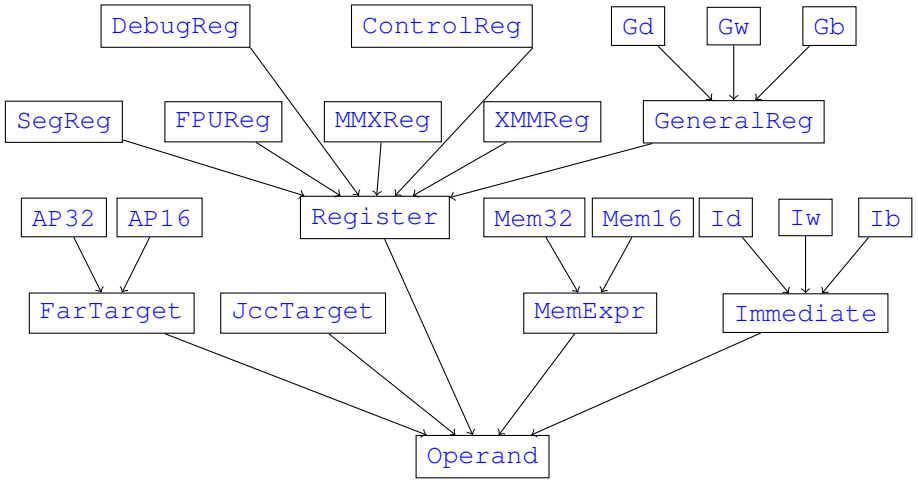
We also employ the concept of a `GuardedInteger`. We have already seen an example of this concept in the introductory section on class properties, section 1.5.5. We want to specify how large a particular type of integer may be, and truncate any values that exceed the size of the integer by masking it with an appropriate constant (e.g., `x & 0xFF` for an 8-bit integer, `x & 0xFFFF` for a 16-bit integer, etc.).

2.2 Representing Instructions and Operands: X86.py

We will begin by discussing our X86 library's interface, as provided in `X86.py`. As before, we have dispensed with overriding the boring Python **`object`** utility methods and have banished them into a set of base classes in `X86Internal.py` (using the techniques from section **??**).

2.2.1 Representing X86 Operands

Previously, in section **??**, we divided our bit-vector expressions (`BVExpr`s) into categories by having them derive from common base classes. This benefitted the structure of other bit-vector modules, as in the type-checker in section **??**. We use a similar grouping to organize our class hierarchy for X86 objects, as follows:



Five classes derive directly from `Operand`; we examine each in turn in the following subsections. The interfaces for these classes are summarized in table 4, and explored in detail in subsequent sections.

1. `Register`
2. `Immediate`
3. `FarTarget`
4. `MemExpr`
5. `JccTarget`

Table 4: `Operand` Classes and their Data Members

`Register` derivatives: `Gb`, `Gw`, `Gd`, `ControlReg`, `DebugReg`, `SegReg`, `FPUReg`, `MMXReg`, `XMMReg`	
`Register`	`value`: a enumeration element of the proper type `IntValue()`: returns the register's integer value `__call__(i)`: creates a new `Register` object of the proper type corresponding to register #`i` in the associated enumeration `__init__(i,adjust_value=False)`: if `adjust_value` is `False`, `i` is interpreted as an enumeration element; otherwise, it is interpreted as a register number, and converted to an enumeration element.
`Immediate` derivatives: `Ib`, `Iw`, `Id`	
`Immediate`	`value`: a guarded integer of the proper size `__call__(n)`: creates a new `Immediate` object of the proper type with the value `n`
`JccTarget`	`_taken`: a guarded integer object for the taken address `_nottaken`: a guarded integer object for the taken address
`FarTarget` derivatives: `AP16`, `AP32`	
`FarTarget`	`Seg`: a guarded 16-bit integer `Off`: a guarded 16 or 32-bit integer
`MemExpr` derivatives: `Mem16`, `Mem32`	
`MemExpr`	`Seg`: a guarded 16-bit integer `size`: a memory size enumeration element `BaseReg`: a guarded 16 or 32-bit base register (may be `None`) `IndexReg`: a guarded 16 or 32-bit index register (may be `None`) `Disp`: a guarded 16 or 32-bit integer (may be `None`) `__call__(s)`: creates a duplicate of the `MemExpr` object with the segment `s`
`Mem32`	`ScaleFac`: a guarded 2-bit integer for the scale factor

2.2.2 X86 Registers: `Register`

The X86 instruction set defines many registers. We collect them by having them all derive from the common base class `Register`. This and other tricks simplify the X86 encoder and decoder.

Category	Size	Values
`Gb`	8	`al`, `cl`, `dl`, `bl`, `ah`, `ch`, `dh`, `bh`
`Gw`	16	`ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si`, `di`
`Gd`	32	`eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`
`SegReg`	16	`es`, `cs`, `ss`, `ds`, `fs`, `gs`
`ControlReg`	32	`cr0`, `cr1`, `cr2`, `cr3`, `cr4`, `cr5`, `cr6`, `cr7`
`DebugReg`	32	`dr0`, `dr1`, `dr2`, `dr3`, `dr4`, `dr5`, `dr6`, `dr7`
`FPUReg`	80	`st0`, `st1`, `st2`, `st3`, `st4`, `st5`, `st6`, `st7`
`MMXReg`	64	`mm0`, `mm1`, `mm2`, `mm3`, `mm4`, `mm5`, `mm6`, `mm7`
`XMMReg`	128	`xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6`, `xmm7`

For example, to create the register `al`, use the Python code `Gb(Al)`.

In X86 machine code, registers are often represented by an associated integer 0-7 (in the corresponding order in which they are listed in the table above). We take this into account in the design of the `Register` class to simplify encoding and decoding.

- When encoding instructions, we will need to retrieve the integer for a given `Register` object. We accomplish this with a `Register` class method `IntValue()`, which returns the number.
- When decoding instructions, we will need to create the correct `Register` object for a given integer. It would be convenient to be able to create register objects simply by passing a number to the constructor. By default, `Register` object constructors accept an enumeration element such as `Al`. We achieve our desires of convenience using an extra parameter in the `Register` constructor: `__init__(self,value,adjust_value=False)`. If `adjust_value` is `True`, the parameter `value` is interpreted as an integer, and is adjusted into an enumeration element by adding the base element of the enum. For example, constructing the object `Gb(1,True)` yields `Gb(Cl)`.
- We use a further trick to simplify decoding. We shall see later that, when it is time to decode a register operand, we shall already have a `Register`-derived object of the correct type available to us (let's call that object `o`). We will want to create a new object of the same type with the correct register value corresponding to an integer obtained during decoding. We override in `Register` one of Python's built-in **`object`** methods, `__call__(self,int)`, which allows objects to be *called*, as in `o(1)`. Our `__call__` implementation constructs a new object of the same type as `o` with the register value specified by the integer **`int`**. While decoding, we can simply "call" `o` with that integer to create the proper `Register` object. For example, with `o = Gb(Al)`, the code snippet `o(1)` yields `Gb(Cl)`. This is an instance of a **factory method** design pattern in software engineering.

2.2.3 X86 Immediates: `Immediate`

Category	Size	Values	Example
`Ib`	8	`12h`, `0FFh`	`Ib(0x12)`
`Iw`	16	`1234h`, `0FFFFh`	`Iw(0x123)`
`Id`	32	`12345678h`, `0FFFFFFFFh`	`Id(0x12345)`

Immediates are stored internally in a field called `value`. This a guarded integer object, as previously discussed in 2.1. I.e., upon construction (and in case of modification), the integer `value` is truncated to the proper size.

For reasons similar to those described in the previous section 2.2.2, we simplify the decoder by overriding Python **`object`**'s `__call__` method. Given an object `i` derived from `Immediate` (as illustrated in the table), we can call `i(n)` to obtain a new `Immediate` object of the same type as `i`, whose integer value is initialized to `n`.

2.2.4 X86 Jump Targets: JccTarget

```
.text:00401024 jnz loc_401050   JccTarget(0x401250,0x401026)
.text:00401024 call loc_401050  JccTarget(0x401250,0x401026)
.text:00401024 jmp loc_401050   JccTarget(0x401250,0x401026)
```

The JccTarget class is not very noteworthy, except to say that it is not a perfect fit for the call instruction. Consider call eax, represented as Instruction([],Call,Gd(Eax)), and notice that the return address is not represented within the instruction. We address this issue after instruction decoding, in section 2.12.1.

2.2.5 X86 Far Targets: FarTarget

```
call 1234h:12345678h  AP32(0x1234,0x12345678)
jmp 1234h:5678h       AP16(0x1234,0x5678)
```

The classes AP32 and AP16 hold their data members (the numeric segment and offset) in guarded integers, which are accessible through the class properties Seg and Off. AP32 uses 32-bit offsets, while AP16 uses 16-bit offsets. Both use 16-bit segments.

2.2.6 X86 Memory Expressions: MemExpr

Memory expressions are the most complex operands to represent. We need classes for both 16- and 32-bit memory accesses; we have them derive from a common base class. The standard Python boilerplate **object** methods for memory operands are defined in the class MemExpr, from which Mem16 and Mem32 derive. MemExpr also holds data elements common to both Mem16 and Mem32 memory expressions:

- Seg: the enumeration element corresponding to the segment into which the access is taking place.
- size: the enumeration element corresponding to the size of the access. Mb designates 8-bit, Mw 16-bit, Md 32-bit, etc.
- BaseReg: the base register for the expression (may be None).
- IndexReg: the index register for the expression (may be None).
- Disp: a displacement (differs in size for 16 vs. 32-bit expressions, and may be None).
- The class method DefaultSeg() returns the segment that would normally be used for this particular combination of base register and displacement, if no segment prefixes were specified.
- As with registers and immediates, the class method __call__(self,seg) returns a new object of the correct type (i.e., Mem16 or Mem32), which is an exact duplicate of the current class instance self, except that it uses the segment seg rather than the current value of self.Seg.

`Mem32` also exposes a data item (class property) called `ScaleFac`. MOD R/M-16 expressions cannot have scale factors, so this data was not placed into the common `MemExpr` class. `ScaleFac` is a 2-bit integer denoting the scale factor (such as `4` in the expression `[eax+ebx*4]`). In the table below, note that a scale factor of `1 << x` is represented by `x`.

`ScaleFac` (binary)	Scale Factor
00	1
01	2
10	4
11	8

2.3 Byte Streams: X86ByteStream.py

Our X86 library has a nice feature not found in all such libraries. When decoding instructions, the only information required by the decoder is an address and access to bytes at that address. It does not actually matter where the bytes come from – whether the source is a disk file, memory, a database (as in IDA), or anything else. It is advantageous to design our library with this idea in mind. Such flexibility allows our library to be used in a variety of circumstances, for example:

- We can run our code inside of IDA and use the IDA API function `get_byte()` as a byte stream (as in section **??**).
- We can use a PE-parsing library and provide bytes from an executable file.
- We can use our code in conjunction with a debugger library, and use whatever facilities it offers to read bytes out of memory.

The instruction and MOD R/M decoders shall be written in such a way where they obtain bytes as needed from a **byte stream object**. `X86ByteStream.py` defines the interface in a class called `StreamObj`, whose methods are shown in table 5. Any input source that can be accessed in terms of a class derived from `StreamObj` can be used seamlessly within our framework; such classes need only override `GetByteInternal()`, which defaults to retrieving a byte from an array (called `bytes`) held within the object.

One note about the `StreamObj` class is that, since X86-32 instructions cannot exceed 16 bytes in length, `Byte()` will throw an `InvalidInstruction` exception if more than 16 bytes have been consumed since the last call to `SetPos()` (and `Word()` and `Dword()` are implemented in terms of `Byte()`, so they too can throw exceptions).

2.4 Testing the X86 Library

Ultimately, we will write a type-checker, an instruction encoder, and a decoder. These components are fairly complex and it would be nice to have some degree of confidence that our code works correctly. Fortuitously, our situation has the property that, if we encode an instruction `instr`, we should be able to decode it and obtain that same instruction `instr`. Encoding an instruction requires type-checking, so we can be sure that

Table 5: `StreamObj` Class Members

Class Method	Description
`Init()`	Sets position-related variables to zero.
`SetPos()`	Sets the position within the byte stream.
`Pos()`	Retrieves the position of the byte stream.
`GetByteInternal()`	Retrieves a byte from the source.
`Byte()`	Retrieves a byte.
`Word()`	Retrieves a little-endian word.
`Dword()`	Retrieves a little-endian dword.

all three components work and their results agree for that particular instruction. We will create a random instruction generator in section 2.8, and thus we can ensure test coverage for all components on all parts of the instruction set. Our X86 library is of very high quality.

Two notes. First, the reader might have the idea that the reverse process could also work for testing: generate random bytes, decode them, encode them, and check that the bytes are the same. This does not work well because some instructions can be encoded in multiple different ways. Secondly, note that, while our method of testing achieves high code coverage and gives us reasonable confidence in our library, in fact, the tests are not checking for "correctness" of our library per se. The tests ensure that the encoder and decoder are **consistent**. Our tests will not detect situations where the encoding and decoding components both contain the same error (i.e., instructions are encoded erroneously, and decoded in the same erroneous way).

2.5 MOD R/M: X86ModRM.py

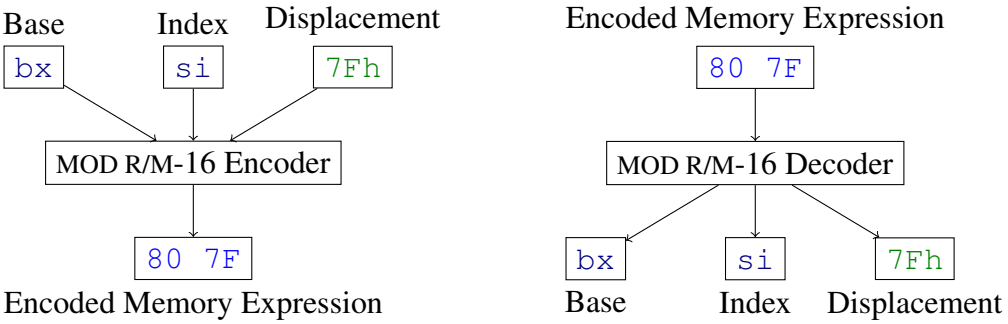
X86 MOD R/M, two binary coding schemes for memory expressions, are instrumental to encoding and decoding X86 machine code. We require functionality to convert memory expressions into bytes encoded with these schemes, and also to perform the inverse process of reconstructing memory expressions from the binary representations. When an X86 CPU is operating in 32-bit mode, memory expressions are 32-bits by default, i.e. encoded using MOD R/M-32. The ADDRSIZE prefix causes a 16-bit memory expression, and MOD R/M-16 memory expression encoding, to be used instead. The next two sections describe the code for encoding and decoding MOD R/M-16 and MOD R/M-32 expressions.

2.5.1 MOD R/M-16

In figure 7, we represent the MOD R/M-16 encoding and decoding components as black boxes. The encoder takes the components of a memory expression and produces bytes representing them. The decoder takes bytes representing a MOD R/M-16-encoded memory expression, and returns the components of the memory expression. You, the student, shall be responsible for implementing encoding and decoding.

Turning your attention to `X86ModRM.py`, this module begins by defining the class `ModRM16`. It exports the following data items (via class properties), corresponding to the parts of the MOD R/M encoding:

Figure 7: MOD R/M-16 Encoding and Decoding as Black Boxes



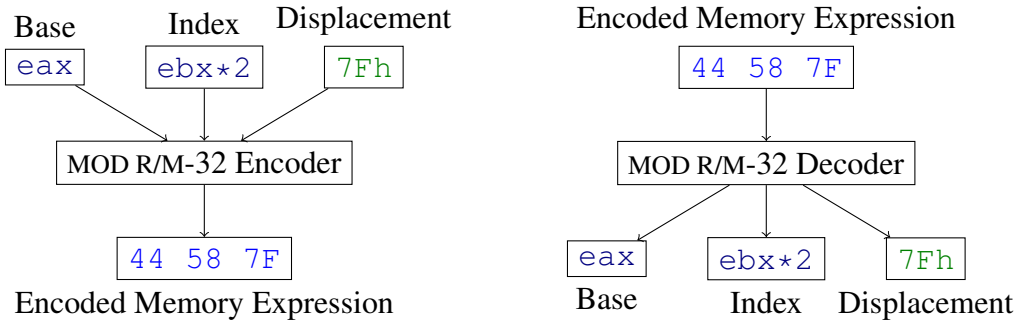
- `MOD`, a 2-bit guarded integer.
- `GGG`, a 3-bit guarded integer.
- `RM`, a 3-bit guarded integer.
- `Disp`, a 16-bit guarded integer.
- `DispSize`, an integer describing the size of the displacement in bytes.

It provides several class methods:

- `Decode(self,stream)`, which decodes the raw MOD R/M bytes provided by the `stream` object (as in section 2.3), and sets the fields listed just previously.
- `Interpret(self)`, which examines the sub-parts of the MOD R/M and returns a quadruple `(base,index,disp,dispsize)`, where:
  - `base` is the base register for the memory expression (or `None`).
  - `index` is its index register (or `None`).
  - `disp` is its displacement (or `None`).
  - `dispsize` is the number of bytes in the displacement (`0`, `1`, or `2`).
- `Encode(self)`, which encodes the MOD R/M parts into a list of bytes.
- `EncodeAdditional(self)`, encodes any "additional" information required by the MOD R/M. In the `ModRM16` class (the one we are discussing presently), it does nothing. This method will be overridden when `ModRM32` derives from `ModRM16`, so we will discuss it again later.
- `EncodeFromParts(self,base,index,disp)`, which examines the parts of the memory expression passed in as parameters, and sets the internal fields corresponding to MOD R/M parts accordingly. Those arguments have the same meaning as described for the return value of the `Interpret` method above.

Exercise: See exercises 6, 5, and 7 in the exercise manual.

Figure 9: MOD R/M-32 Encoding and Decoding as Black Boxes



2.5.2 MOD R/M-32

Treatment of MOD R/M-32 is very similar to MOD R/M-16. In fact, the class `ModRM32` derives from `ModRM16`, so it offers the same class methods and members. All data members have the same meaning as described in the previous subsection, and the class methods have virtually identical interfaces in both classes.

MOD R/M-32 allows scaled components within memory expressions (e.g., `ebx*2` in `[eax+ebx*2]`), while MOD R/M-16 did not. The slides discussed how these components are encoded ysing so-called SCALE-INDEX-BASE, or **SIB**, byte. Similarly to how MOD R/M is divided into three fields, so is the SIB byte. We represent SIB components as a class, `SIBBase`, which holds three class properties `SCALE`, `INDEX`, and `BASE`. The `ModRM32` class contains an instance of the `SIBBase` class, held in a class property named `SIB`, which is set to `None` if a SIB is not required.

To accomodate for scaled components, two class methods have slightly different interfaces than in `ModRM16`.

- `Interpret(self)`, which returns a pentuple `(base,index,scale,disp,dispsize)`, whereas the method in `ModRM16` returned a quadruple (since the `scale` component was not necessary).
- `EncodeFromParts(self,base,index,scale,disp)`, which takes an additional parameter `scale`.

Exercise: See exercises 9, 8, and 10 in the exercise manual.

2.6 The Encoding Table: X86EncodeTable.py

The slides discussed **instruction encodings** for a mnemonic, as can be seen for example in table 6. These are defined in `X86EncodeTable.py`. An encoding consists of:

1. An **encoding method** (Ordinary, Size Prefix, or MOD R/M Group), represented by classes derived from `X86Enc` that are respectively named `Ordinary`, `Native16`, and `ModRMGroup`.
2. An **instruction stem**, member `stem`, represented as a list of bytes.

Table 6: All Encodings for `add`

Encoding Method	Stem Bytes	Abstract Operand Types	Python Representation
Ordinary	00	OEb OGb	`Ordinary([0x00],[OEb,OGb])`
Ordinary	01	OEv OGv	`Ordinary([0x01],[OEv,OGv])`
Ordinary	02	OGb OEb	`Ordinary([0x02],[OGb,OEb])`
Ordinary	03	OGv OEv	`Ordinary([0x03],[OGv,OEv])`
Ordinary	04	OAL OIb	`Ordinary([0x04],[OAL,OIb])`
Ordinary	05	OrAX OIz	`Ordinary([0x05],[OrAX,OIz])`
MOD R/M Group #0	80	OEb OIb	`ModRMGroup(0,[0x80],[OEb,OIb])`
MOD R/M Group #0	81	OEv OIz	`ModRMGroup(0,[0x81],[OEv,OIz])`
MOD R/M Group #0	83	OEv OIbv	`ModRMGroup(0,[0x83],[OEv,OIbv])`

3. A list of **abstract operand types**, member `ops`, represented by enumeration elements defined in `X86InternalOperand.py`.

The `X86Enc` objects contain all data associated with a particular instruction encoding. They also expose a class method called `Encode` that is invoked during encoding. We shall ignore this method for now and return to it when discussing encoding.

2.7 Abstract Operand Types and their Descriptions

`X86InternalOperand.py` enumerates the abstract operand types defined in the Intel Manuals' opcode maps. Additionally, as described in the slides, we represent each abstract operand type with its description in a small language we called AOTDL. This very simple language is defined in `X86InternalOperandDescriptions.py`. The base class is `X86AOTDL`, and its derived classes' data members are shown in table 7.

Table 7: `X86AOTDL` Classes and their Data Members

`Exact`	`value`: an object derived from `Operand`
`ExactSeg`	`value`: an object derived from `MemExpr`
`ImmEnc`	`archetype`: an X86 operand object. May be derived from `Imm`, `MemExpr`, `FarTarget`, or `JccTarget`.
`SignedImm`	`archetype`: an object derived from `Imm` or `JccTarget`.
`GPart`	`archetype`: an object derived from `Register`
`RegOrMem`	`reg`: an object derived from `Register` or `None` `mem`: a memory size (i.e., `Mb`) or `None`
`SizePrefix`	`yes`: AOTDL object derived from `X86AOTDL` (prefix present) `no`: AOTDL object derived from `X86AOTDL` (prefix absent)
`AddrPrefix`	`yes`: AOTDL object derived from `X86AOTDL` (prefix present) `no`: AOTDL object derived from `X86AOTDL` (prefix absent)

Following the declaration of the classes for AOTDL is an array, `AOTtoAOTDL`, which assigns an element of this language to each abstract operand type. Table 9 shows some examples. The right-hand columns are the actual Python objects that are used to represent the abstract operand types on the left.

2.8 Random Generation of X86 Objects: X86Random.py

In section 2.4, we mentioned that we shall test our X86 library using random instruction generation to ensure high code coverage and high quality. We begin with random operand generation. This serves as a convenient moment to reinforce why AOTDL is such a useful mechanism, and we take the time to walk through a simple, yet complete example of how to use it. `X86Random.py` begins with declarations of helper functions such as `rnd_bool()`, `rnd_dword()`, etc. for generating random numbers of specified sizes. Before generating a random operand for a given abstract operand type, we create three random boolean values:

- `m`: if the operand type could potentially be either a register or a memory expression, try to use a memory expression if this value is `True`.
- `s`: if the operand type varies with the OPSIZE prefix, use the smaller size if this variable is `True`.
- `a`: if the operand type varies with the ADDRSIZE prefix, use the smaller size if this variable is `True`.

To actually generate a random operand, we retrieve the AOTDL description for the operand type, and use pattern-matching to decide which logic should execute. The pattern-matching rules and associated random operand generation logic, the function `generate(a)`, is shown in table 8. The reader who feels confident with pattern-matching should skip the following bulleted list and read the table instead.

- The AOTDL element `Exact(o)` specifies that `o` is the only legal value for some abstract operand type. Therefore, to "randomly generate" a legal element of this abstract operand type, we only have one choice: `o` itself. We return `o` as-is.
- `ExactSeg(o)` specifies that the only legal value for some abstract operand type is a memory expression `o`, whose segment may vary. Therefore, our choices in random generation are the exact copies of the memory expression `o`, with different segments. Recall from section 2.2.6 that we overrode `__call__(seg)` in the `MemExpr` base class in order to produce a new memory expression that is an exact duplicate of the current one, except that its segment is set to `seg`. Thus, we can simply "call" `o`, as in `return o(rnd_seg())`.
- `GPart(o)` AOTDL elements specify registers that have the same type as the parameter `o`. Recall from section 2.2.2 that we overrode `__call__(int)` in the `Register` base class in order to produce a new register of the same type with the register enumeration element corresponding to the integer `int`. Most register types have eight possible values, so they can be handled with the same logic, namely `return o(rnd_regno())`. Register types with fewer than eight values, such as `SegReg`, are handled individually.
- `RegOrMem(r,m)` AOTDL elements can either specify registers or memory expressions. We consult our randomly-generated boolean `m` to determine whether to to

generate a register or a memory. Register types are handled identically as just described under `GPart`. For memory expressions, if the randomly-generated boolean `a` is specified, we generate a `Mem16` object with random components; otherwise, we generate a `Mem32` object.

- `ImmEnc(o)` AOTDL elements can describe immediate constants, memory expressions, segment:offset addresses, or jump targets. We must inspect `o`'s type to determine which case applies.
  - If `o` is an X86 `Immediate` object (such as `Ib(0x12)`), we need to return an object of the same type (e.g., `Ib`) with a randomly-generated constant value. Recall from section 2.2.3 that we overrode `__call__(value)` in `Immediate` in order to produce a new constant of the same type with value `value`. Hence, we can simply return `o(rnd_dword())`. If the constant type is below 32-bits in size, its constructor will truncate the value to the correct number of bits automatically.
  - If `o` is a `MemExpr`, `ImmEnc(o)` corresponds to a memory expression that consists solely of an address (i.e., no base or index registers). Depending upon the value of our randomly-generated boolean `a`, we may generate either a `Mem16` or a `Mem32` object.
  - If `o` is a `FarTarget`, `ImmEnc(o)` corresponds to a segment:offset memory location. Depending upon the value of our randomly-generated boolean `a`, we may generate either an `AP16` or an `AP32` object.
  - If `o` is a `JccTarget`, return a `JccTarget` with randomly-generated taken and not-taken addresses.
- `SignedImm(o)` AOTDL elements describe 8-bit constant values that are sign-extended to larger values. `o` can describe immediate constants or jump targets. We must inspect `o`'s type to determine which case applies.
  - If `o` is an X86 `Immediate` object (such as `Iw(0x0)`), we generate a random 8-bit constant and sign-extend it to the larger size. As before, having overriden `Immediate.__call__(value)` simplifies the task.
  - If `o` is a `JccTarget`, return a `JccTarget` with randomly-generated taken and not-taken addresses.
- `SizePrefix(t,f)` AOTDL elements behave differently depending upon the OPSIZE prefix, i.e. our randomly-generated boolean `s`. We invoke `generate` on either `t` or `f` depending upon the value of `s`.
- `AddrPrefix(t,f)` AOTDL elements behave differently depending upon the ADDRSIZE prefix, i.e. our randomly-generated boolean `a`. We invoke `generate` on either `t` or `f` depending upon the value of `a`.

Implementing random operand generation is easy with the visitor pattern. Our previous visitor classes have all derived from `Visitor`, which was the generic visitor implementation for when the visit method should take one operand. This algorithm uses

Table 8: Random Operand Generation: `generate(a)`

AOTDL `a`	Condition	Random generation code
`Exact(o)`		`return o`
`ExactSeg(o)`		`return o(rnd_seg())`
`GPart(SegReg(s))`		`return SegReg(rnd_seg())`
`GPart(r)`		`return r(rnd_regno())`
`RegOrMem(r,m)`	`m and a`	`return rnd_mem16(m)`
`RegOrMem(r,m)`	`m`	`return rnd_mem32(m)`
`RegOrMem(r,m)`		`return r(rnd_regno())`
`ImmEnc(Immediate(i))`		`return i(rnd_dword())`
`ImmEnc(MemExpr)`	`a`	`return Mem16(rnd_seg(),None,None,rnd_word())`
`ImmEnc(MemExpr)`		`return Mem32(rnd_seg(),None,None,0,rnd_dword())`
`ImmEnc(FarTarget)`	`a`	`return AP16(rnd_word(),rnd_word())`
`ImmEnc(FarTarget)`		`return AP32(rnd_word(),rnd_dword())`
`ImmEnc(JccTarget)`		`return JccTarget(rnd_dword(),rnd_dword())`
`SignedImm(Imm(i))`		`return i(sign_extend(rnd_byte()))`
`SignedImm(JccTarget)`		`return JccTarget(rnd_dword(),rnd_dword())`
`SizePrefix(t,f)`	`s`	`return generate(t)`
`SizePrefix(t,f)`		`return generate(f)`
`AddrPrefix(t,f)`	`a`	`return generate(t)`
`AddrPrefix(t,f)`		`return generate(f)`

two parameters: the abstract operand type, and the triple of booleans `(m,s,a)`. Thus we derive our random operand generator, `X86RandomOperand`, from `Visitor2` rather than `Visitor`. We override the `MakeMethodName` method to produce names such that roughly every row of the table has one single method responsible for implementing it. These methods have names like `visit_Immediate_MemExpr`, `visit_Immediate_FarTarget`, and `visit_GPart_SegReg`.

Clients of the random operand generator should call the method `gen(aop,(m,s,a))`, which takes care of retrieving the `X86AOTDL` object associated with the abstract operand type `aop` prior to calling `visit`. The logic in the `visit_` methods is more or less exactly as described in the table.

Generating random instructions is just as simple.

1. Pick a random X86 mnemonic enumeration element, `mnem`, and retrieve its list of encodings from the encoder table.
2. Choose a random encoding, and retrieve its list of abstract operand type enumeration elements, `aotlist`.
3. Generate three random booleans, `m`, `s`, and `a`, as above in operand generation.
4. Generate random operands for each element of `aotlist`.
5. Return `Instruction([],mnem,op1,op2,op3)`.

Figure 11: `typecheck_x86(a,x)`

a	x	Type-Checking Logic	Return Value
`Exact(o)`	`x`	`x == o`	MATCHES
`ExactSeg(o)`	`MemExpr(_)`	`x == o`	MATCHES
`ExactSeg(o)`	`MemExpr(_)`	`x == o(x.Seg)`	`SegPFX(x.Seg)`
`GPart(g)`	`x`	`type(g) == type(x)`	MATCHES
`RegOrMem(r,m)`	`Register(y)`	`type(r) == type(y)`	MATCHES
`RegOrMem(r,m)`	`Mem32(_)`	`m.size == x.size`	`AddrPFX(False)`
`RegOrMem(r,m)`	`Mem16(_)`	`m.size == x.size`	`AddrPFX(True)`
`ImmEnc(Immediate(i))`	`Immediate(y)`	`type(i) == type(y)`	MATCHES
`ImmEnc(JccTarget(_))`	`JccTarget(_)`		MATCHES
`ImmEnc(FarTarget(_))`	`AP16(_)`		`AddrPFX(False)`
`ImmEnc(FarTarget(_))`	`AP32(_)`		`AddrPFX(True)`
`ImmEnc(MemExpr(_))`	`Mem32(_)`	`x.BaseReg == None and x.IndexReg == None`	`AddrPFX(False)`
`ImmEnc(MemExpr(_))`	`Mem16(_)`	`x.BaseReg == None and x.IndexReg == None`	`AddrPFX(True)`
`SignedImm(JccTarget(_))`	`JccTarget(_)`		MATCHES
`SignedImm(Id(_))`	`Id(i)`	`0xFFFFFF80 <= i <= 0x7F`	MATCHES
`SignedImm(Iw(_))`	`Iw(i)`	`0xFF80 <= i <= 0x7F`	MATCHES

2.9 Type-Checking: X86TypeChecker.py

Thanks to AOTDL, type-checking is a simple affair. As in the previous section 2.8, the algorithm is implemented by deriving from `Visitor2` and having one class method for each type-checking rule (with the `Visitor2` method `MakeMethodName` overridden to create the proper method names). The type-checking logic is replicated from the slides in figure 11. If this table intimidates you, please re-read the extended example in the previous section 2.8.

The type-checking information for a given operand is represented by a class called `TypeCheckInfo`. Type-checking an operand against an abstract operand type yields either a `TypeCheckInfo` object, or `None` if it did not match. `TypeCheckInfo` contains three data items:

- `sizeo`: if `None`, this operand is not affected by the presence of the OPSIZE prefix. If `True` or `False`, the operand respectively requires the presence or absence of the OPSIZE prefix.
- `addro`: behaves identically to `sizeo`, except for the ADDRSIZE prefix.
- `sego`: is `None` if no segment prefix is required, and a segment enumeration element otherwise.

We define a few convenience functions to simplify the job. The function `SizePFX(b,t=None)` either updates an existing `TypeCheckInfo` object `t`'s `sizeo` member to the boolean `b`, or, if `t` is `None`, returns a new `TypeCheckInfo` object with `sizeo=b`. `AddrPFX(b,t=None)` and `SizePFX(s,t=None)` behave similarly, except their operations respectively concern the `addro` and `sego` members of the `TypeCheckInfo` object.

When the type-checker encounters the AOTDL elements `SizePrefix(yes,no)` and `AddrPrefix(yes,no)`, the operand could match either possibility (the `yes` or `no` AOTDL element). Therefore, we must check the operand `x` against both `yes` and `no`. If it matches `yes`, we return a value indicating that the prefix was possible, and also that it was required. If it matches `no`, we return a value indicating that the prefix was possible, but that it was not required. If neither matches, return `None`. The pseudocode below shows the basic logic for `SizePrefix`; `AddrPrefix` behaves identically, with `SizePFX` replaced by `AddrPFX`.

`typecheck_x86(a,x)`, for `a = SizePrefix(p,n)`

```
# Check prefix required AOTDL
yr = typecheck_x86(a.yes,x)
if yr is not None:
  return SizePFX(True, yr)

# Check prefix absent AOTDL
nr = typecheck_x86(a.no,x)
if nr is not None:
  return SizePFX(False,nr)

return None
```

All of the valid type-checking rules are listed figure 11, apart from the `SizePrefix` and `AddrPrefix` AOTDL elements just discussed. Any other combination of operands and operand types is illegal. They will be routed to the `Visitor2` method `Default`, which we override to return a value indicating that there was no match. The client should call the method `check(aop,opnd)`, where `aop` is an abstract operand type enumeration element, and `opnd` is an X86 operand derived from `Operand`.

Some abstract operand types (AOTs) and their AOTDL translations are shown in table 9.

2.9.1 Type-Checking Instructions

To check an X86 instruction against an encoding (as opposed to checking an X86 operand against an abstract operand type), we:

- Ensure the number of operands in the encoding matches those in the instruction.
- Type-check each instruction operand against each abstract operand.
- Ensure mutual compatibility of the operands, as described in the slides and implemented in `reduce_typeinfo`. E.g., though the operands of `xor eax, bx` match `OGv` and `OEv` individually, the first requires that the OPSIZE prefix be absent, whereas the second requires it be present. This situation is erroneous, so `reduce_typeinfo` throws an exception.

`TypeCheckInstruction_exn(instr,aops)` checks the operands of the instruction `instr` against the list of abstract operands `aops`. If they do not match, it throws the exception `X86TypeCheckError` with a string describing the failure. If they match, it returns a triple `(s,a,seg)`, where:

Table 9: AOTDL Example Translations

AOT	AOTDL
OAL	`Exact(Gb(Al))`
OYw	`Exact(Mem32(ES,Mw,Edi,None,0,None))`
OXb	`ExactSeg(Mem32(DS,Mb,Esi,None,0,None))`
OIw	`ImmEnc(Iw)`
OId	`ImmEnc(Id)`
OOb	`ImmEnc(MemExpr(DS,Mb))`
OGb	`GPart(Gb)`
OGw	`GPart(Gw)`
OGd	`GPart(Gd)`
OEb	`RegOrMem(Gb,Mb)`
ORdMb	`RegOrMem(Gd,Mb)`
OStN	`RegOrMem(FPUReg(ST0),None)`
OeAX	`SizePrefix(Exact(Gw(Ax)),Exact(Gd(Eax)))`
OGv	`SizePrefix(GPart(Gw),GPart(Gd))`
OIv	`SizePrefix(ImmEnc(Iw),ImmEnc(Id))`
OIbv	`SizePrefix(SignedImm(Iw),SignedImm(Id))`
OEv	`SizePrefix(RegOrMem(Gw,Mw),RegOrMem(Gd,Md))`
OM	`AddrPrefix(RegOrMem(None,Mw),RegOrMem(None,Md))`
OYb	`AddrPrefix(Exact(Mem16(ES,Mb,Di,None,None)),` `Exact(Mem32(ES,Mb,Edi,None,0,None))`
OXw	`AddrPrefix(ExactSeg(Mem16(DS,Mw,Si,None,None)),` `ExactSeg(Mem32(DS,Mw,Esi,None,0,None))`

- `s` is a boolean dictating whether an OPSIZE prefix is required.
- `a` is a boolean dictating whether an ADDRSIZE prefix is required.
- `seg` is a segment enumeration element for a segment prefix, or `None` if no such prefix is required.

There are two interface methods provided for type-checking instructions: `TypeCheckInstruction_exn` as just described, and `TypeCheckInstruction_opt`. The latter calls the former inside of a **`try`**/**`except`** block, and returns `None` if an exception was thrown. The suffixes `_exn` and `_opt` evidence this distinction in functionality: the former throws an exception, the latter returns an optional value (i.e., one that may be `None`).

Exercise: See exercises 11 and 12 in the exercise manual.

2.10 Encoding: X86Encoder.py

AOTDL pays off again handsomely in developing the X86 encoder. `X86Encoder.py` declares the `X86Encoder` class, which is what the slides called the **encoder context**, and whose interface is shown in table 10.

Table 10: `X86Encoder` Class

`group1pfx`	A prefix enumeration element or `None`.
`segpfx`	A segment enumeration element or `None`.
`sizepfx`	A boolean: whether an OPSIZE prefix is required.
`addrpfx`	A boolean: whether an ADDRSIZE prefix is required.
`stem`	A list of bytes (initially empty) for the instruction stem.
`ModRM`	A MOD R/M object as a class property, initially `None`.
`immediates`	A list of bytes (initially empty) for immediate operands.
`Reset()`	Sets the variables above to their default values.
`addr`	The integer address at which to encode.
`tc`	An `X86TypeChecker` object.
`AppendImmediate(imm,n)`	Append the `n`-byte, little-endian integer `imm` to `immediates`.
`EncodeInstruction(instr,addr=0)`	Find a suitable encoding for `instr`, and encode it. Return the encoded bytes as a list. Raise an exception if none is found.
`EncodeInstructions(instrs,addr=0)`	Encode multiple instructions using `EncodeInstruction`.

The `EncodeInstruction` class method is the interface through which clients encode single instructions. It is a very direct implementation of the instruction encoding process as laid out in the slides.

1. Retrieve the mnemonic's list of encodings from the `mnem_to_encodings` table in `X86EncodeTable.py`.

2. For each encoding, consult the `X86TypeChecker` object `tc` held within the encoder context as to whether the instruction matches it. If a valid encoding `enc` is found, retrieve the required prefixes, store them into the variables held encoder context, and copy `enc`'s stem bytes.

3. Address any special requirements of the encoding method, by invoking `enc.Encode(self)`. The `enc` object may be of type `Ordinary`, `Native16`, or `ModRMGroup`. The bodies of these classes' `Encode` methods are shown below.

`X86Enc.Encode` Methods from `X86EncodeTable.py`

```
class Ordinary(X86Enc):
  pass

class Native16(X86Enc):
  def Encode(self,enc):
    enc.sizepfx = True

class ModRMGroup(X86Enc):
  def Encode(self,enc):
    enc.ModRM.GGG = self.ggg
```

4. Encode the operands. This is implemented on the `X86Encoder` class, as usual, in terms of a visitor pattern. Class method `visit(x86op,aop)` inspects the types

of `x86op` and the `X86AOTDL` entry corresponding to `aop`, and invokes a specialized function for each case. The encoding rules are duplicated from the slides in figure 12.

5. Concatenate the instruction parts, yielding X86 machine code. At this stage in the encoding process, all of the information has been collected in order to encode the instruction. We simply concatenate the prefixes, stem, optional MOD R/M, and optional immediate operand bytes, and return them as a list.

Figure 12: `encode_x86(a,x)`

a	x	Encoder Logic
`Exact(o)`	`_`	
`ExactSeg(o)`	`_`	
`GPart(g)`	`_`	`ModRM.GGG = x.IntValue()`
`RegOrMem(r,m)`	`Register(y)`	`ModRM.RM = y.IntValue()` `ModRM.MOD = 3`
`RegOrMem(r,m)`	`Mem32(_)`	`EncodeModRM32(x)`
`RegOrMem(r,m)`	`Mem16(_)`	`EncodeModRM16(x)`
`ImmEnc(Immediate(_))`	`Id(i)`	`immediates.append(i,4)`
`ImmEnc(Immediate(_))`	`Iw(i)`	`immediates.append(i,2)`
`ImmEnc(FarTarget(_))`	`AP32(s,o)`	`immediates.append(s,2)` `immediates.append(o,4)`
`ImmEnc(FarTarget(_))`	`AP16(s,o)`	`immediates.append(s,2)` `immediates.append(o,2)`
`ImmEnc(MemExpr(_))`	`Mem32(_)`	`immediates.append(x.Disp,4)`
`ImmEnc(MemExpr(_))`	`Mem16(_)`	`immediates.append(x.Disp,2)`
`SignedImm(Immediate(_))`	`_`	`immediates.append(x.value,2)`
`ImmEnc(JccTarget(t,f))`	`JccTarget(_)`	`immediates.append(t - next_addr,4)`
`SignedImm(JccTarget(t,f))`	`JccTarget(_)`	`immediates.append(t - next_addr,4)`
`SizePrefix(y,n)`	`_`	`if sizepfx: encode_x86(y,x)` `else: encode_x86(n,x)`
`AddrPrefix(y,n)`	`_`	`if addrpfx: encode_x86(y,x)` `else: encode_x86(n,x)`

2.10.1 Seamless Handling of MOD R/M

Some X86 instructions require MOD R/M as part of their encoding; some do not. Class properties allow us to handle this situation in a very elegant way. The actual `ModRM16` or `ModRM32` object is held in a `X86Encoder` class member called `_modrm`. Before encoding, `_modrm`. is set to `None`, to indicate that a MOD R/M is not required. During encoding, such as when encoding the operands, a MOD R/M object might become necessary. It would be ugly to clutter the code with checks to ensure that `_modrm` is not `None` and to allocate an object otherwise.

Instead, client code accesses `_modrm` via a class property called `ModRM`, as in code such as `enc.ModRM.GGG = 3`. The getter property for `ModRM` (the class method `ModRM(self)`) checks to see whether `_modrm` is `None`. If it is not, a MOD R/M object has already been allocated, so it returns `_modrm`. If it is `None`, the getter property allocates a `ModRM16` or

ModRM32 object (depending upon addrpfx, i.e. whether a ADDRSIZE prefix is required) on the fly, stores it in _modrm, and returns that to the client. Creation of MOD R/M objects is handled entirely transparently, meaning that the client can write code such as enc.ModRM.GGG = 3 without worrying about whether a MOD R/M object has already been allocated.

Exercise: See exercises 13 and 14 in the exercise manual.

2.11 The Decoder Table: X86DecodeTable.py

Figure 13: Intel Opcode Maps, 00-87

	0	1	2	3	4	5	6	7
0	ADD						PUSH	POP
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	ES	ES
1	ADC						PUSH	POP
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SS	SS
2	AND						SEG=ES	DAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	(Prefix)	
3	XOR						SEG=SS	AAA
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	(Prefix)	
4	INC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
5	PUSH general register							
	rAX	rCX	rDX	rBX	rSP	rBP	rSI	rDI
6	PUSHA/ PUSHAD	POPA/ POPAD	BOUND Gv, Ma	ARPL Ew, Gw	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc, Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv

At the heart of our X86 decoder engine is a representation of the opcode maps from the Intel manuals, as partially shown in figure 13. In general, when writing code that deals with X86 machine code, it is best to represent the data in a way that is similar to the way it is laid out in the manuals. This aids faithfullness to the manuals and easy verification that the transcribed data matches them, and allows easy maintainability. In addition, the pervasive use of tables makes the code easier to write and harder to get wrong.

The slides laid out a simple description language for representing the entries of opcode maps. X86DecodeTable.py contains the Python class declarations for the DECDL decoder table description language, all of which derive from X86DECDL. The language, and our treatment of it, it is very similar to AOTDL and its accompanying classes as previously described in 2.7. Those classes and their data members are shown in table 11.

In addition to the data members, each of these classes exports a method called decode(decoder). The discussion of these methods is left to the next section.

After the declarations of these classes, the bulk of X86DecodeTable.py is the array decoding_table, of length 1024. It contains one such X86DECDL object per instruction stem.

Table 11: X86DECDL Classes and their Data Members

Direct(m,o)	m: X86 mnemonic enumeration element o: list of abstract operand type enumeration elements
PredMOD(m,r)	m: X86DECDL for when MOD R/M.MOD selects memory r: X86DECDL for when MOD R/M.MOD selects a register
PredOpSize(y,n)	y: X86DECDL for when OPSIZE is present n: X86DECDL for when OPSIZE is absent
PredAddrSize(y,n)	y: X86DECDL for when OPSIZE is present n: X86DECDL for when OPSIZE is absent
Group(l)	l: list of 8 X86DECDL entries, one per MOD R/M.GGG
RMGroup(l)	l: list of 8 X86DECDL entries, one per MOD R/M.RM
SSE(n,r,s,z)	n: X86DECDL for when no prefixes match r: X86DECDL for the rep prefix s: X86DECDL for the OPSIZE prefix z: X86DECDL for the repnz prefix
Invalid	

2.12 Decoding: X86Decoder.py

X86Decoder.py contains the X86 disassembler functionality, implemented in a class called X86Decoder. Its class members and data items are shown in table 12. The constructor requires a byte stream object, as discussed in section 2.3, from which to consume bytes while decoding.

Table 12: X86Decoder Class

group1pfx	A list of prefix enumeration elements, initially empty.
segpfx	A segment enumeration element or None.
sizepfx	A boolean: whether an OPSIZE prefix is required.
addrpfx	A boolean: whether an ADDRSIZE prefix is required.
ModRM	A MOD R/M object as a class property, initially None.
Reset()	Sets the variables above to their default values.
stream	An X86ByteStream object, from which to consume bytes.
GetSegment()	Returns the segment that should be used for a memory expression.
DecodePrefixes()	Consume the prefixes from the byte stream; update the internal variables accordingly. Return the first non-prefix byte.
DecodeStem(first_byte)	Consume the stem from the byte stream and return it.
Decode(ea)	Decode and return an instruction at address ea.

The X86Decoder class method Decode(ea) implements the decoding algorithm as described in the slides to decode an instruction located at address ea. It calls a series of class methods in order to do so, which shall be described subsequently.

1. Consume the prefixes. Decode(ea) begins by invoking the class method DecodePrefixes That method retrieves a byte from the stream object, updates the relevant class members of X86Decoder if it was a prefix byte, and repeats. When the first non-prefix byte is encountered, return that byte.

2. Consume the stem. X86 employs so-called **escape bytes** as part of its variable-length instruction encoding scheme. Instruction stems may begin with `0F`, `0F 38`, or `0F 3A`. The class method `DecodeStem(first_byte)` consumes up to two more bytes from the `stream` object, and returns an integer from `00-FF` for instructions with no escape bytes, from `100-1FF` for instructions that began with `0F`, from `200-2FF` for instructions that began with `0F 38`, and from `300-3FF` for instructions that began with `0F 3A`.

3. Retrieve decoder entry from decoder table. Given the stem returned previously, use it as an index into `decoding_table` from `X86DecodeTable.py`. This gives us an `X86DECDL` object.

4. Process decoder entry. Table 13 shows the logic for decoding DECDL entries. This logic is implemented within the `decode` method of the `X86DECDL` classes. Most of the DECDL entries simply query the state of the `X86Decoder` object and call the `decode` method on some `X86DECDL` object held within. The `X86DECDL` class `Invalid` raises an `InvalidInstruction` exception. `Direct(m,o)` `X86DECDL` objects correspond to instructions with mnemonic `m` and list of abstract operand type enumeration elements `o`. In this case, we return `m,o`.

5. Decode operands. If we have reached this point in the code, we have a mnemonic and a list of abstract operand types. As usual, operand decoding is implemented via a visitor pattern within the `X86Decoder` class. We invoke `visit` on each abstract operand type in the list `o` to create the actual X86 `Operand` objects. The decoding rules are shown in figure 14.

6. Return an `Instruction` object. In actuality, we return slightly more than this; we take it up in the next subsection.

Table 13: Decoding Logic for `X86DECDL` Decoder Entries

DECDL	Condition	Return Value
`Direct(m,o)`		`(m,o)`
`Invalid`		`raise InvalidInstruction`
`PredMOD(m,r)`	`ModRM.MOD != 3`	`m.decode()`
`PredMOD(m,r)`		`r.decode()`
`PredOpSize(y,n)`	`sizepfx`	`y.decode()`
`PredOpSize(y,n)`		`n.decode()`
`PredAddrSize(y,n)`	`addrpfx`	`y.decode()`
`PredAddrSize(y,n)`		`n.decode()`
`Group(l)`		`l[ModRM.GGG].decode()`
`RMGroup(l)`		`l[ModRM.RM].decode()`
`SSE(n,s,r,z)`		(Complex logic)

As was the case during encoding, some X86 instructions require a MOD R/M component, and some do not. We handle this situation exactly as we did in the encoder. The MOD R/M object is held in a class member called `_modrm`, which is initially set to `None`.

Figure 14: `decode(a)`

a	Condition	Return Value
`Exact(o)`		`return o`
`ExactSeg(o)`	`segpfx == None`	`return o`
`ExactSeg(o)`		`return o(segpfx)`
`ImmEnc(Immediate(Id))`		`return Id(Dword())`
`ImmEnc(Immediate(Iw))`		`return Iw(Word())`
`ImmEnc(Immediate(Ib))`		`return Ib(Byte())`
`ImmEnc(FarTarget(_))`	`addrpfx`	`return AP16(Word(),Word())`
`ImmEnc(FarTarget(_))`		`return AP32(Word(),Dword())`
`SignedImm(Id(_))`		`return Id(sign_extend(Byte()))`
`SignedImm(Iw(_))`		`return Iw(sign_extend(Byte()))`
`SizePrefix(p,n)`	`sizepfx`	`return decode(p)`
`SizePrefix(p,n)`		`return decode(n)`
`AddrPrefix(p,n)`	`addrpfx`	`return decode(p)`
`AddrPrefix(p,n)`		`return decode(n)`
`ImmEnc(MemExpr(m))`	`addrpfx`	`return Mem16(segpfx,m.Size,None,None,Word())`
`ImmEnc(MemExpr(m))`		`return Mem32(segpfx,m.Size,None,None,0,Dword())`
`GPart(g)`		`return g(ModRM.GGG)`
`RegOrMem(r,m)`	`ModRM.MOD == 3`	`return r(ModRM.RM)`
`RegOrMem(r,m)`	`addrpfx`	`return DecodeModRM16(m.size)`
`RegOrMem(r,m)`		`return DecodeModRM32(m.size)`

`ImmEnc(JccTarget(_))`		`displ = Dword()`
		`nextaddr = stream.ea()`
		`return JccTarget(nextaddr+displ,nextaddr)`

`SignedImm(JccTarget(_))`		`displ = sign_extend(Byte())`
		`nextaddr = stream.ea()`
		`return JccTarget(nextaddr+displ,nextaddr)`

Client code accesses the MOD R/M information via a class property, `ModRM`. Whenever the instruction decoding code references `ModRM`, the getter method checks to see whether `_modrm` is `None`, and returns `_modrm` directly if not. If `_modrm` was `None`, the getter decodes either a MOD R/M-16 or MOD R/M-32 depending upon the value of `addrpfx`, i.e. whether an ADDRSIZE prefix was consumed during prefix decoding, stores it into `_modrm`, and returns it. As before, this allows for seamless handling of MOD R/M.

Exercise: See exercises 15 and 16 in the exercise manual.

2.12.1 Flow Information: ASMFlow.py

After an instruction has been decoded, we package it up with one additional class, `X86DecodedInstruction` in `X86.py`. This class holds an instruction, the address at which the instruction began, the length of the instruction, and the flow information for the instruction. The flow information describes to which other instructions this instruction may transfer execution, in a format that is useful for reasoning about control flow in other parts of the framework. This information can be used, for example, to write more sophisticated disassemblers like IDA's, which employs **recursive traversal** to explore entire control flow graphs.

Flow information is stored in a utility file, `ASMFlow.py`. Its classes are described in

table 14. To create this information, we simply inspect an instruction's mnemonic and operand types after decoding and create the required class.

Table 14: Flow Information Classes and Their Data Members

`FlowOrdinary(nextaddr)`	`nextaddr`: next address
`FlowJmpUnconditional(target)`	`target`: jump target
`FlowJmpConditional(taken,nottaken)`	`taken`: jump taken address `nottaken`: jump not taken address
`FlowCallDirect(calldest,retaddr)`	`calldest`: destination address `retaddr`: return address
`FlowCallIndirect(retaddr)`	`retaddr`: return address
`FlowJmpIndirect`	
`FlowReturn`	

After decoding an instruction, we inspect its mnemonic and operands to determine its control flow characteristics. This logic is shown in table 15, and implemented in class method `CreateFlow`.

Table 15: `CreateFlow` Logic

Mnemonic	Operand	Flow Type
`call`	`JccTarget(t,f)`	`FlowCallDirect(t,f)`
`call`	`GeneralReg(_)`	`FlowCallIndirect(retaddr)`
`call`	`FarTarget(_,_)`	`FlowCallIndirect(retaddr)`
`jmp`	`JccTarget(t,_)`	`FlowJmpUnconditional(t)`
`jmp`	`GeneralReg(_)`	`FlowJmpIndirect`
`jmp`	`FarTarget(_,_)`	`FlowJmpIndirect`
`jo`, `jno`, `jb`, `jae`, `jz`, `jnz`, `jbe`, `ja`, `js`, `jns`, `jp`, `jnp`, `jl`, `jge`, `jle`, `jg`, `loopnz`, `loopz`, `loop`, `jcxz`, `jecxz`	`JccTarget(t,f)`	`FlowJmpConditional(t,f)`
`ret`, `retf`, `iretd`, `iretw`	N/A	`FlowReturn`
Anything Else	N/A	`FlowOrdinary(nextaddr)`

3 Reference Material

3.1 X86

3.1.1 MOD R/M-16

r8			AL	CL	DL	BL	AH	CH	DH	BH
r16			AX	CX	DX	BX	SP	BP	SI	DI
r32			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
ymm			YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
sreg			ES	CS	SS	DS	FS	GS		
ccc			CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
ddd			DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7
digit			0	1	2	3	4	5	6	7
ggg			000	001	010	011	100	101	110	111
effective address	mod	R/M	value of mod R/M byte (hex)							
[BX+SI]	00	000	00	08	10	18	20	28	30	38
[BX+DI]		001	01	09	11	19	21	29	31	39
[BP+SI]		010	02	0A	12	1A	22	2A	32	3A
[BP+DI]		011	03	0B	13	1B	23	2B	33	3B
[SI]		100	04	0C	14	1C	24	2C	34	3C
[DI]		101	05	0D	15	1D	25	2D	35	3D
[sword]		110	06	0E	16	1E	26	2E	36	3E
[BX]		111	07	0F	17	1F	27	2F	37	3F
[BX+SI+sbyte]	01	000	40	48	50	58	60	68	70	78
[BX+DI+sbyte]		001	41	49	51	59	61	69	71	79
[BP+SI+sbyte]		010	42	4A	52	5A	62	6A	72	7A
[BP+DI+sbyte]		011	43	4B	53	5B	63	6B	73	7B
[SI+sbyte]		100	44	4C	54	5C	64	6C	74	7C
[DI+sbyte]		101	45	4D	55	5D	65	6D	75	7D
[BP+sbyte]		110	46	4E	56	5E	66	6E	76	7E
[BX+sbyte]		111	47	4F	57	5F	67	6F	77	7F
[BX+SI+sword]	10	000	80	88	90	98	A0	A8	B0	B8
[BX+DI+sword]		001	81	89	91	99	A1	A9	B1	B9
[BP+SI+sword]		010	82	8A	92	9A	A2	AA	B2	BA
[BP+DI+sword]		011	83	8B	93	9B	A3	AB	B3	BB
[SI+sword]		100	84	8C	94	9C	A4	AC	B4	BC
[DI+sword]		101	85	8D	95	9D	A5	AD	B5	BD
[BP+sword]		110	86	8E	96	9E	A6	AE	B6	BE
[BX+sword]		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/MM0/XMM0/YMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/MM1/XMM1/YMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/MM2/XMM2/YMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/MM3/XMM3/YMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/MM4/XMM4/YMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/MM5/XMM5/YMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/MM6/XMM6/YMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/MM7/XMM7/YMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

3.1.2 MOD R/M-32

r8			AL	CL	DL	BL	AH	CH	DH	BH
r16			AX	CX	DX	BX	SP	BP	SI	DI
r32			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
ymm			YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
sreg			ES	CS	SS	DS	FS	GS		
eee			CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
eee			DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7
digit			0	1	2	3	4	5	6	7
reg=			000	001	010	011	100	101	110	111
effective address	mod	R/M	value of mod R/M byte (hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[sib]		100	04	0C	14	1C	24	2C	34	3C
[sdword]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX+sbyte]	01	000	40	48	50	58	60	68	70	78
[ECX+sbyte]		001	41	49	51	59	61	69	71	79
[EDX+sbyte]		010	42	4A	52	5A	62	6A	72	7A
[EBX+sbyte]		011	43	4B	53	5B	63	6B	73	7B
[sib+sbyte]		100	44	4C	54	5C	64	6C	74	7C
[EBP+sbyte]		101	45	4D	55	5D	65	6D	75	7D
[ESI+sbyte]		110	46	4E	56	5E	66	6E	76	7E
[EDI+sbyte]		111	47	4F	57	5F	67	6F	77	7F
[EAX+sdword]	10	000	80	88	90	98	A0	A8	B0	B8
[ECX+sdword]		001	81	89	91	99	A1	A9	B1	B9
[EDX+sdword]		010	82	8A	92	9A	A2	AA	B2	BA
[EBX+sdword]		011	83	8B	93	9B	A3	AB	B3	BB
[sib+sdword]		100	84	8C	94	9C	A4	AC	B4	BC
[EBP+sdword]		101	85	8D	95	9D	A5	AD	B5	BD
[ESI+sdword]		110	86	8E	96	9E	A6	AE	B6	BE
[EDI+sdword]		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/MM0/XMM0/YMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/MM1/XMM1/YMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/MM2/XMM2/YMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/MM3/XMM3/YMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
AH/SP/ESP/MM4/XMM4/YMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
CH/BP/EBP/MM5/XMM5/YMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
DH/SI/ESI/MM6/XMM6/YMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
BH/DI/EDI/MM7/XMM7/YMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

3.1.3 MOD R/M-32 SIB

r32			EAX	ECX	EDX	EBX	ESP	[*]	ESI	EDI
digit			0	1	2	3	4	5	6	7
base=			000	001	010	011	100	101	110	111
scaled index	SS	Index	value of SIB byte (hex)							
[EAX*1]	00	000	00	01	02	03	04	05	06	07
[ECX*1]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX*1]		010	10	11	12	13	14	15	16	17
[EBX*1]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP*1]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI*1]		110	30	31	32	33	34	35	36	37
[EDI*1]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	99	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

3.1.4 AOTDL

Language Element					Meaning
`aotdl`	::=	`Exact`	`x86op`		`x86op` exactly.
	|	`ExactSeg`	`x86op`		`x86op` memory expression whose segment may vary.
	|	`Immediate`	`x86op`		Immediate of type `x86op`.
	|	`SignedImm`	`x86op`		Sign-extended immediate of type `x86op`.
	|	`GPart`	`regtype`		Register from family `regtype`.
	|	`ModRM`	`regtype`	`memsize`	Register `regtype` or memory `memsize`. Either may be `None`.
	|	`SizePrefix`	`aotdl`	`aotdl`	`aotdl` with and without OPSIZE.
	|	`AddrPrefix`	`aotdl`	`aotdl`	`aotdl` with and without ADDRSIZE.

3.1.5 DECDL

Language Element					Meaning
decdl	≔	Direct	mnem	[aotdl]	Mnemnonic mnem, operands [aotdl].
	|	Invalid			Illegal instruction.
	|	PredMOD	decdl	decdl	decdl for MOD ≠ 3 and MOD = 3.
	|	PredOpSize	decdl	decdl	decdl with and without OPSIZE.
	|	PredAddrSize	decdl	decdl	decdl with and without ADDRSIZE.
	|	Group	[decdl]*8		One decdl entry for each MOD R/M.GGG.
	|	RMGroup	[decdl]*8		One decdl entry for each MOD R/M.RM.
	|	SSE	[decdl]*4		One decdl entry for each SSE prefix.