• Tuples are immutable sequences.
• Dictionaries are unordered collections of key-value pairs.

Dictionary keys do have two restrictions:
• A key of a dictionary cannot be an object of a mutable built-in type.
• Two keys cannot be equal. There can be at most one value for a key.

Lists are mutable sequences.
Tuples are immutable sequences.

A range is a sequence of consecutive integers.

```python
>>> range(-2, 2)
Range([-2, -1, 0, 1, 2])
```

Length. A sequence has a finite length.

Element selection. A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

A function with a parent frame

Every call changes the balance

Python pre-computes which frame contains each name before executing the body of a function. Therefore, within the body of a function, all instances of a name must refer to the same frame.

```
def make_withdraw(balance):
    def withdraw(amount):
        if amount > balance:
            raise InsufficientFunds
        balance -= amount
        return balance
    return withdraw

withdraw = make_withdraw(100)
withdraw(25)
withdraw(25)
withdraw(25)
```

Mutable values can be changed without a nonlocal statement.

UnboundLocalError: local variable 'balance' referenced before assignment

Identity testing is performed by "is" and "is not" operators. Binding an object to a new name using assignment does not create a new object:

```python
>>> a = Account('Jack')
>>> b = Account('Jack')
```

The def statement header is similar to other functions

Conditional statements check

Base cases are evaluated without recursive calls

Typically, all other cases are evaluated with recursive calls.

A new object:

```python
>>> b = Account('Joker')
```

For each element in that sequence, in order:
- `for` iterable value.
- `in` iterable value.

```python
for <name> in <expression>:
    <suite>
```

1. Evaluate the header `<expression>`, which must yield an iterable value.
2. For each element in that sequence, in order:
   A. Bind `<name>` to that element in the local environment.
   B. Execute the `<suite>`.

List comprehensions

```python
[<map exp> for <name> in <iter exp> if <filter exp>]
```

Short version: `[<map exp> for <name> in <iter exp>]

Unlike generator expressions, the map expression is evaluated when the list comprehension is evaluated.

```python
>>> suits = ['heart', 'diamond', 'spade', 'club']
>>> from unicodedata import lookup
>>> [lookup('WHITE CARD', s.upper()) + ' SUIT' for s in suits]
[('♤', '♧', '♢', '♦')]
```

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jack')
>>> b = Account('Jack')
```

Identity testing is performed by "is" and "is not" operators. Binding an object to a new name using assignment does not create a new object:

```python
>>> a = Account('Jack')
>>> a is a
True
>>> a is not b
True
```

`List comprehensions`

```python
[<map exp> for <name> in <iter exp> if <filter exp>]
```

Short version: `[<map exp> for <name> in <iter exp>]

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

The def statement header is similar to other functions

Conditional statements check

Base cases are evaluated without recursive calls

Typically, all other cases are evaluated with recursive calls

```python
def pig_latin(w):
    return w[1:] + w[0]
```

```python
def starts_with_a_vowel(w):
    return w[0].lower() in 'aeiou'
```

```python
def pig_latin(w):
    return w[1:] + w[0]
```
**A class statement creates a new class and binds that class to a name in the first frame of the current environment.**

2. **Statements in the suite create attributes of the class.**

To evaluate a dot expression: `<expression> . <name>`
1. Evaluate the `<expression>` to the left of the dot, which yields the object of the dot expression.
2. `<name>` is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If not, `<name>` is looked up in the class, which yields a class attribute value.
4. That value is returned unless it is a function, in which case a bound method is returned instead.

To look up a name in a class:
1. If it names an attribute in the class, return the attribute value.
2. Otherwise, look up the name in the base class, if there is one.

When a class is called:
1. A new instance of that class is created.
2. The constructor `__init__` of the class is called with the new object as its first argument (called `self`), along with additional arguments provided in the call expression.

```python
class Account(object):
    __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

def deposit(self, amount):
    self.balance += amount
    return self.balance

def withdraw(self, amount):
    if amount > self.balance:
        return 'Insufficient funds'
    self.balance -= amount
    return self.balance

>>> a = Account('Jim')
```

**A mutable Rlist implementation using message passing**

```python
def mutable_rlist():
    contents = empty_rlist
    dispatch(message, value=None):
        if message == 'len':
            return len_rlist(contents)
        elif message == 'getitem':
            return getitem_rlist(contents, value)
        elif message == 'push_first':
            contents = make_rlist(value, contents)
            f = first(contents)
            contents = rest(contents)
            return f
        elif message == 'str':
            return str(contents)
    return dispatch
```

**A bank account implemented using dispatch dictionaries**

```python
def account(balance):  return 'Insufficient funds'
    dispatch['balance'] = amount
    return dispatch['balance']

def deposit(amount):
    dispatch['balance'] = amount
    return dispatch['balance']

def withdraw(amount):
    if amount > dispatch['balance']:
        dispatch['balance'] = amount
        return dispatch['balance']
```

**A simple container implemented using two accessor methods**

```python
def container(contents):
    def get();
        return contents
    def put(value):
        return put, get
```

**Type dispatching**: Define a different function for each possible combination of types for which an operation is valid.

```python
def complex(z):
    return type(z) in (ComplexRI, ComplexMA)
def isreal(z):
    return type(z) == Real
def add_complex_and_rational(z, r):
    return ComplexR(z.real + r.real, z.imag + r.imag)
def add_by_typedispatch(z, r):
    if iscomplex(z) and iscomplex(r):
        return add_complex_and_rational(z, r)
```

1. Try to coerce arguments into values of the same type
2. Apply type-specific (not cross-type) operations

```python
def coerce_apply(operator_name, x, y):
    if ty = type_tag(x).type_tag(y):
        if x = ty:
            return operator(x, r)
        elif ty in coercions:
            tx = ty coercion[(tx, ty)](x)
        else:
            return 'No coercion possible.'
```