Mutability

Class outline:

- Objects & methods
- List mutation & methods
- Tuples
- Mutability
- Beware of mutation

Objects

Objects

An **object** is a bundle of data and behavior.

A type of object is called a **class**.

Every value in Python is an object.

- All objects have attributes
- Objects often have associated methods

Strings as objects

```
name = 'PamelamaDingDong'
```

What kind of object is it?

```
type(name)
```

What data is inside it?

```
name[0]
name[8:]
```

What methods can we call?

```
name.upper()
name.lower()
```

List mutation

Mutating lists with methods

`append()` adds a single element to a list:

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t = 0
```

Try in PythonTutor.

`extend()` adds all the elements in one list to a list:

```
s = [2, 3]
t = [5, 6]
s.extend(4)
s.extend(t)
t = 0
```

Try in PythonTutor.

Mutating lists with methods

`append()` adds a single element to a list:

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t = 0
```

Try in PythonTutor.

`extend()` adds all the elements in one list to a list:

```
s = [2, 3]
t = [5, 6]
s.extend(4) #  Error: 4 is not an iterable!
s.extend(t)
t = 0
```

Try in PythonTutor. (After deleting the bad line)

Mutating lists with methods

`pop()` removes and returns the last element:

```
s = [2, 3]
t = [5, 6]
t = s.pop()
```

Try in PythonTutor.

`remove()` removes the first element equal to the argument:

```
s = [6, 2, 4, 8, 4]
s.remove(4)
```

Try in PythonTutor.

Mutating lists with slicing

We can do a lot with just brackets/slice notation:

```
L = [1, 2, 3, 4, 5]

L[2] = 6

L[1:3] = [9, 8]

L[2:4] = []              # Deleting elements

L[1:1] = [2, 3, 4, 5]    # Inserting elements

L[len(L):] = [10, 11]    # Appending

L = L + [20, 30]
```

Try in PythonTutor.

Dictionary mutation

Dictionary mutation

Starting with an empty dict:

```
users = {}
```

Add values:

```
users["profpamela"] = "b3stp@ssEvErDontHackMe"
```

Change values:

```
users["profpamela"] += "itsLongerSoItsMoreSecure!!"
```

```
>>> users["profpamela"]
```

Dictionary mutation

Starting with an empty dict:

```
users = {}
```

Add values:

```
users["profpamela"] = "b3stp@ssEvErDontHackMe"
```

Change values:

```
users["profpamela"] += "itsLongerSoItsMoreSecure!!"
```

```
>>> users["profpamela"]
'b3stp@ssEvErDontHackMeitsLongerSoItsMoreSecure!!'
```

Tuples

Tuples

A **tuple** is an immutable sequence. It's like a list, but no mutation allowed!

An empty tuple:

```
empty = ()
# or
empty = tuple()
```

A tuple with multiple elements:

```
conditions = ('rain', 'shine')
# or
conditions = 'rain', 'shine'
```

A tuple with a single element: 😿

Tuples

A **tuple** is an immutable sequence. It's like a list, but no mutation allowed!

An empty tuple:

```
empty = ()
# or
empty = tuple()
```

A tuple with multiple elements:

```
conditions = ('rain', 'shine')
# or
conditions = 'rain', 'shine'
```

A tuple with a single element: 😿

```
oogly = (61,)
# or
oogly = 61,
```

Tuple operations

Many of list's read-only operations work on tuples.

Combining tuples into a new tuple:

```
('come', '☂') + ('or', '☼')
```

Checking containment:

```
'wally' in ('wall-e', 'wallace', 'waldo')
```

Slicing:

```
rainbow = ('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'v
roy = rainbow[:3]
```

Tuple operations

Many of list's read-only operations work on tuples.

Combining tuples into a new tuple:

```
('come', '☂') + ('or', '☼')  # ('come', '☂', 'or', '☼')
```

Checking containment:

```
'wally' in ('wall-e', 'wallace', 'waldo')
```

Slicing:

```
rainbow = ('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'v
roy = rainbow[:3]
```

Tuple operations

Many of list's read-only operations work on tuples.

Combining tuples into a new tuple:

```
('come', '☂') + ('or', '☼')  # ('come', '☂', 'or', '☼')
```

Checking containment:

```
'wally' in ('wall-e', 'wallace', 'waldo')  # True
```

Slicing:

```
rainbow = ('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'v
roy = rainbow[:3]
```

Tuple operations

Many of list's read-only operations work on tuples.

Combining tuples into a new tuple:

```
('come', '☂') + ('or', '☼')  # ('come', '☂', 'or', '☼')
```

Checking containment:

```
'wally' in ('wall-e', 'wallace', 'waldo')  # True
```

Slicing:

```
rainbow = ('red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'v
roy = rainbow[:3]  # ('red', 'orange', 'yellow')
```

Immutability vs. Mutability

Immutable vs. Mutable

An **immutable** value is unchanging once created.

Immutable types (that we've covered): int, float, string, tuple

```
a_tuple = (1, 2)
a_tuple[0] = 3
a_string = "Hi y'all"
a_string[1] = "I"
a_string += ", how you doing?"
an_int = 20
an_int += 2
```

A **mutable** value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.

Mutable types (that we've covered): list, dict

```
grades = [90, 70, 85]
grades_copy = grades
grades[1] = 100
words = {"agua": "water"}
words["pavo"] = "turkey"
```

Immutable vs. Mutable

An **immutable** value is unchanging once created.

Immutable types (that we've covered): int, float, string, tuple

```
a_tuple = (1, 2)
a_tuple[0] = 3                  #  Error! Tuple items cannot be set.
a_string = "Hi y'all"
a_string[1] = "I"               #  Error! String elements cannot be set.
a_string += ", how you doing?"
an_int = 20
an_int += 2
```

A **mutable** value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.

Mutable types (that we've covered): list, dict

```
grades = [90, 70, 85]
grades_copy = grades
grades[1] = 100
words = {"agua": "water"}
words["pavo"] = "turkey"
```

Immutable vs. Mutable

An **immutable** value is unchanging once created.

Immutable types (that we've covered): int, float, string, tuple

```
a_tuple = (1, 2)
a_tuple[0] = 3                  #  Error! Tuple items cannot be set.
a_string = "Hi y'all"
a_string[1] = "I"               #  Error! String elements cannot be set.
a_string += ", how you doing?"  #  How does this work?
an_int = 20
an_int += 2                     #  And this?
```

A **mutable** value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.

Mutable types (that we've covered): list, dict

```
grades = [90, 70, 85]
grades_copy = grades
grades[1] = 100
words = {"agua": "water"}
words["pavo"] = "turkey"
```

Name change vs. mutation

The value of an expression can change due to either changes in names or mutations in objects.

Name change:

```
x + x
```

```
x + x
```

Object mutation:

```
x + x
```

```
x + x
```

Name change vs. mutation

The value of an expression can change due to either changes in names or mutations in objects.

Name change:

```
x = 2
x + x # 4


x + x
```

Object mutation:

```

x + x


x + x
```

Name change vs. mutation

The value of an expression can change due to either changes in names or mutations in objects.

Name change:

```
x = 2
x + x # 4

x = 3
x + x # 6
```

Object mutation:

```
x + x


x + x
```

Name change vs. mutation

The value of an expression can change due to either changes in names or mutations in objects.

Name change:

```
x = 2
x + x # 4

x = 3
x + x # 6
```

Object mutation:

```
x = ['A', 'B']
x + x  # ['A', 'B', 'A', 'B']


x + x
```

Name change vs. mutation

The value of an expression can change due to either changes in names or mutations in objects.

Name change:

```
x = 2
x + x # 4

x = 3
x + x # 6
```

Object mutation:

```
x = ['A', 'B']
x + x  # ['A', 'B', 'A', 'B']

x.append('C')
x + x  # ['A', 'B', 'C', 'A', 'B', 'C']
```

Mutables inside immutables

An immutable sequence may still change if it contains a mutable value as an element.

```
t = (1, [2, 3])
t[1][0] = 99
t[1][1] = "Problems"
```

Try in PythonTutor

Equality of contents vs. Identity of objects

```
list1 = [1,2,3]
list2 = [1,2,3]
```

Equality: `exp0 == exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to objects containing equal values

```
list1 == list2
```

Equality of contents vs. Identity of objects

```
list1 = [1,2,3]
list2 = [1,2,3]
```

Equality: `exp0 == exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to objects containing equal values

```
list1 == list2  # True
```

Equality of contents vs. Identity of objects

```
list1 = [1,2,3]
list2 = [1,2,3]
```

Equality: `exp0 == exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to objects containing equal values

```
list1 == list2  # True
```

Identity: `exp0 is exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to the same object

Identical objects always have equal values.

```
list1 is list2
```

Try in PythonTutor.

Equality of contents vs. Identity of objects

```
list1 = [1,2,3]
list2 = [1,2,3]
```

Equality: `exp0 == exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to objects containing equal values

```
list1 == list2  # True
```

Identity: `exp0 is exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to the same object

Identical objects always have equal values.

```
list1 is list2  # False
```

Try in PythonTutor.

Beware, Mutation!

Mutation in function calls

An function can change the value of any object in its scope.

```
four = [1, 2, 3, 4]
print(four[0])
do_stuff_to(four)
print(four[0])
```

Try in PythonTutor

Even without arguments:

```
four = [1, 2, 3, 4]
print(four[3])
do_other_stuff()
print(four[3])
```

Try in PythonTutor

Immutability in function calls

Immutable values are protected from mutation.

Tuple

```
turtle = (1, 2, 3)
ooze()
turtle # (1, 2, 3)
```

List

```
turtle = [1, 2, 3]
ooze()
turtle # [1, 2, 'Mwahaha']
```

Mutable default arguments 

A default argument value is part of a function value, not generated by a call.

```
def f(s=[]):
    s.append(3)
    return len(s)

f() # 1
f() # 2
f() # 3
```

Each time the function is called, `s` is bound to the same value.

Mutable functions

A function with changing state

Goal: Use a function to repeatedly withdraw from a bank account that starts with $100.

A function with changing state

Goal: Use a function to repeatedly withdraw from a bank account that starts with $100.

First call to the function:

```
withdraw(25)      # 75
```

A function with changing state

Goal: Use a function to repeatedly withdraw from a bank account that starts with $100.

First call to the function:

```
withdraw(25)      # 75
```

Second call to the function:

```
withdraw(25)      # 50
```

A function with changing state

Goal: Use a function to repeatedly withdraw from a bank account that starts with $100.

First call to the function:

```
withdraw(25)      # 75
```

Second call to the function:

```
withdraw(25)      # 50
```

Third call to the function:

```
withdraw(60)      # 'Insufficient funds'
```

A function with changing state

Goal: Use a function to repeatedly withdraw from a bank account that starts with $100.

What makes it possible?

```
withdraw = make_withdraw_account(100) # Contains a list
```

First call to the function:

```
withdraw(25)      # 75
```

Second call to the function:

```
withdraw(25)      # 50
```

Third call to the function:

```
withdraw(60)      # 'Insufficient funds'
```

Implementing state in functions

A mutable value in the parent frame can maintain the local state for a function.

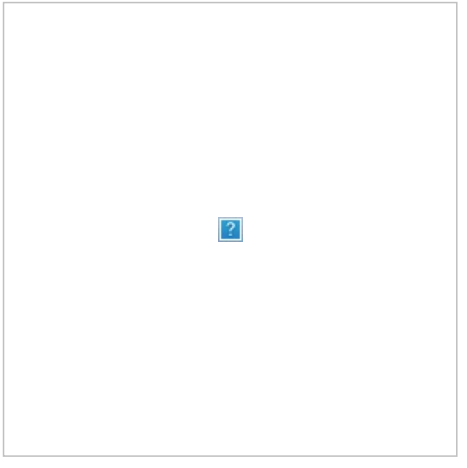
```
def make_withdraw_account(initial):
    balance = [initial]

    def withdraw(amount):
        if balance[0] - amount < 0:
            return 'Insufficient funds'
        balance[0] -= amount
        return balance[0]

    return withdraw
```

View in PythonTutor

Python Project of The Day!

Anki

Anki: An open-source desktop application for studying flash cards.



Technologies used: Python.
(Github repository)