Deeper Dive into Python

Please open this in your browser to follow along:

https://scits.math.unibe.ch/ddip/

Exercises: https://scits.math.unibe.ch/ddip/exercises.zip

Repoistory: https://github.com/scits-bern/ddip

v.1.3 (2021-01-19)

Agenda

What to expect?

What to expect?

This course is aimed at people who either:

1. Know programming coming from another language, and want to see how less-basic language features work.

2. Only ever worked with basics of Python, and want to expand their knowledge.

3. Have survived the "Introduction to Python" course yesterday and understood it well enough.

Please note that I'm not a Python guru. I won't be able to answer all questions about Python without searching the docs or guarantee that my code is "pythonic".

Beyond notebooks

Python (Jupyter) notebooks

If you come from any other Python course we provide, you're likely familiar with Jupyter notebooks.

A notebook is a way to package together snippets ("cells") of Python code, the results of their execution, and arbitrary explanatory text.

It runs as a web application that you normally interact with in a browser.

In case it runs remotely, you don't need to have Python installed on your machine at all.

What are notebooks good at?

- Teaching: minimal setup
- Experimentation: modifying small pieces of code without rerunning everything
- Presentation: can incorporate graphics and explanatory text
- Publishing results: includes output from previous runs of the code

What are notebooks bad at?

- Running non-interactively
- Code modularity and reuse
- Notebooks have confusing semantics (order of execution)
- Environment setup is external to notebooks

What is needed to run Python code?

To run a Python program, you need Python itself installed. If it's installed, you normally can invoke it in a command line interface with `python`.

If you need modules that are not part of the standard library, you use a package manager to install the missing ones; `pip` is the standard one, `conda` is one that comes with Anaconda. More on modules and environments later.

Exercise:

Run `python --version` in a terminal to make sure that Python is installed and ready.

Note:

On systems where multiple versions of python are installed, `python` and `pip` for Python 3.x are sometimes called `python3` and `pip3`

How to run python code?

Python scripts are named with a `.py` extension and can be run with `python code.py` in a terminal.

This executes the whole script at once, unlike cell-by-cell execution of a notebook.

Alternatively, use a code editor that has integrated functionality to run Python code. In case of VSC, you should have a green "run" symbol in the top right corner of the editor when a Python script is open.

Exercise:

Run the sample code `exercises/run_me.py` to make sure your setup works.

Objects and scope

Everything is objects

All data you're working with in Python is represented by objects (or relations between objects).

Examples of objects:

- `None`
- A number, e.g. `1`
- A sequence of objects, e.g. `[1, "two", 3.0, [4]]`
- A function, e.g. `lambda x: x + 1`

Everything is objects

All data you're working with in Python is represented by objects (or relations between objects).

Examples of objects:

- `None`
- A number, e.g. `1`
- A sequence of objects, e.g. `[1, "two", 3.0, [4]]`
- A function, e.g. `lambda x: x + 1`

Some objects are primitive values, some are collections of other objects, some serve a technical purpose (like `None`), etc.

Each object has an *identity*, *type* and *value*.

Object identity

Whenever a new object is created, it is assigned an *identity*.

The identity is a unique integer that:

1. Stays the same within the lifetime of an object,
2. Is different to any other (currently existing) object's identity.

Object identity

Whenever a new object is created, it is assigned an *identity*.

The identity is a unique integer that:

1. Stays the same within the lifetime of an object,
2. Is different to any other (currently existing) object's identity.

```python
a = [1, 2, 3]
b = [1, 2, 3]
c = b
# Query the identity
id(a)  # Returns something like 2041912700496
# Compare identities
a is b # False, despite equal values; equivalent to "id(a) == id(b)"
b is c # True, different names for the same object
```

It is not guaranteed to have any specific meaning, but in CPython implementation it returns the memory location of the object.

The mystery of (small) integers

Exercise:

We saw that assigning the "same" list literal to two variables produces different objects:

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a is b) # Outputs False
```

Repeat this experiment with a large number (e.g. 10000) and a small number (e.g. 5).

Repeat it again in an interactive Python session (call Python without a script to execute).

Can you guess why the results differ?

Object type

Every object has a *type*, assigned at creation time and normally it cannot change.

An object's type is an indication of its structure and operations that are applicable to it.

Built-in type examples:

```
type(None)                 # NoneType
type(1)                    # int
type([1, "two", 3.0, [4]]) # list
type(lambda x: x + 1)      # function
import sys
type(sys)                  # module
```

User-created *classes* provide a way to create new types; we'll get to them soon.

Object value

An object's "value" is an abstract concept and can differ from type to type.

Informally, it's all the data attached to the object.

Object value

An object's "value" is an abstract concept and can differ from type to type.

Informally, it's all the data attached to the object.

In case of a primitive object, its value is obvious, e.g. an `int`'s value is the number it represents.

For a different example, for a list `["a", 1]` one could consider the two objects, in their order, to be the value.

Object value

An object's "value" is an abstract concept and can differ from type to type.

Informally, it's all the data attached to the object.

In case of a primitive object, its value is obvious, e.g. an `int`'s value is the number it represents.

For a different example, for a list `["a", 1]` one could consider the two objects, in their order, to be the value.

Object values may change throughout the lifetime of an object, unlike identity and type.

Mutable vs immutable objects

An object is mutable if its value can change after creation.

Built-in primitive types, like numbers, are immutable: for example, `4 + 1` does not change `4` or `1`, but creates a new object `5`.

Mutable vs immutable objects

An object is mutable if its value can change after creation.

Built-in primitive types, like numbers, are immutable: for example, `4 + 1` does not change `4` or `1`, but creates a new object `5`.

Sometimes, objects carry references to other objects, for example a list `[1, "a"]` is an object that references the object `1` and the object `"a"`.

Some built-in data structures are immutable (e.g. tuples) while other are not (e.g. lists).

If an object's references to other objects can't change, it is considered immutable, even if overall data changes:

```
a = (1, 2, [3, 4])
a[2] = [3, 4, 5] # Produces an error, as tuples are immutable
a[2].append(5) # Works fine: it's still the same list
```

Accessing objects

We usually work with objects accessing them by some name in the scope, e.g. variable names:

```python
a = 100
b = a
# Now "a" and "b" are both names for the same object 100

def increment(x):
    return x + 1
# Now "increment" is a name for that function
```

Note that `x` doesn't mean anything outside that function. This is because `x` is a *local* variable in the function.

Scope

Python has 4 type of scopes, in order in which they are searched:

- *Local* scope, that's created whenever a function is run.
- *Non-local* scope, which contains names from enclosing functions (the closest that matches is used).
- *Global* scope that contains names from the top level of the current module.
- *Builtin* scope that contains Python's built-in names.

You can specify the scope of a variable before use using keywords `local`, `nonlocal`, `global`.

Scope example

```python
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)    # "test spam"
    do_nonlocal()
    print("After nonlocal assignment:", spam) # "nonlocal spam"
    do_global()
    print("After global assignment:", spam)   # "nonlocal spam"

scope_test()
print("In global scope:", spam) # global spam
```

Scope gotchas

If you only read a variable in a function, it will be searched in all scopes. You don't need to declare a variable `global` in a function for it to read a global variable.

However, if at any point in a function a variable is assigned to (including augmented assignment like `x += 1`), it defaults to being local.

```
test = "Something"

def f1():
    print(test)

def f2():
    print(test)
    test = "Something else"

f1() # Outputs "Something"
f2() # Raises UnboundLocalError
```

What can you do with objects?

Objects can have *attributes* (or *properties*): references to other objects accessible by their attribute name.

```
x = 3.0 + 4.0j
x.imag # 4.0 (a property of x)
```

An attribute that's a function is called a *method:*

```
x.conjugate() # 3.0 - 4.0j (a method on x)
```

What else can you do with objects?

There are standard functions and operations that apply to various objects. For example:

```
x + y

len(x)

x[y]

if x:
  pass

for x in y:
  pass
```

What else can you do with objects?

There are standard functions and operations that apply to various objects. For example:

```
x + y

len(x)

x[y]

if x:
  pass

for x in y:
  pass
```

All of those work differently based on the types of objects involved, with some operations having no sense and thus not supported.

Let's learn to make our own objects!

Classes

What are classes?

Classes are user-defined types of Python objects.

A Class is essentially a blueprint of what kind of data needs to be grouped together, along with functions (methods) that make sense for this kind of data.

An object of such type is called a class *instance*, and has its own unique set of data (properties) associated with it.

Classes are a way to have objects that have a defined interface

Code using a class gets a user-defined interface to interact with it, while internal details are not important.

Class syntax

The syntax for declaring a class looks like this:

```python
class ClassName:
    # statement 1
    # ...
    # statement N
```

By convention, class names start with a capital letter.

Most of the time, the statements within the `class` block will be functions for method definitons.

Simplest class

Let's look at the simplest class possible, one that does nothing:

```
class Thing:
    pass
```

Once that definition executes, we can already create instaces of this class by calling the name of the class like a function:

```
something = Thing()
print(something)       # Outputs <__main__.Thing object at 0x000001DB6B754760>
print(type(something)) # Outputs <class '__main__.Thing'>
```

Non-declared goods

We could assign and later access properties on a specific instance, even though the class "blueprint" comes with none:

```
something.prop = "Whatever"
print(something.prop) # Outputs "Whatever"

print(something.prop2)
# AttributeError: 'Thing' object has no attribute 'prop2'
```

In fact, most user-created classes store instance properties in a `dict` structure (the `__dict__` property), and it's possible to add or modify that at runtime.

This can surprise people coming from other programming languages; there are ways to enforce a specific set of properties (with `__slots__`), but it's unusual for Python.

Adding a method

Let's add our first method to a class.

```python
class Greeter:
    def hello(self, target="World"):
        print(f"Hello, {target}!")

g = Greeter()
g.hello() # Outputs "Hello, World!"
g.hello("everyone") # Outputs "Hello, everyone!"
```

You'll notice that the method seems to have one more argument than is actually passed. More on that in a bit.

Aside: f-strings

Requires Python 3.6+

A modern, more readable way to have templated strings, an alternative to `.format()` approach with less boilerplate.

The following code is equivalent:

```
print("The sum of {} + {} is {}".format(x, y, x + y))

print("The sum of {x} + {y} is {sum}".format(x=x, y=y, sum=(x + y)))

print(f"The sum of {x} + {y} is {x + y}")
```

Documentation: Language reference, PEP 498

Class methods

```python
class Greeter:
    def hello(self, target="World"):
        print(f"Hello, {target}!")

g = Greeter()
g.hello("everyone") # Outputs "Hello, everyone!"
```

The first argument of any class method automatically receives the reference to the instance itself.

Put another way, those two calls are equivalent:

```python
g.hello("everyone")
Greeter.hello(g, "everyone")
```

In some programming languages it is called `this` and is often a keyword; in Python, it's just a function argument, which is called `self` by convention.

Using self to store data

We haven't yet used `self` in the code. Let's tell our name if it's defined:

```python
class Greeter:
    def hello(self, target="World"):
        print(f"Hello, {target}!")
        try:
            print(f"My name is {self.name}.")
        except AttributeError:
            print("I don't know my name yet.")

g = Greeter()

g.hello()
# Hello, World!
# I don't know my name yet.

g.name = "Fred"
g.hello("everyone")
# Hello, everyone!
# My name is Fred.
```

Checking for attributes

Instead of using a `try..except` block, we could more gracefully test if an attribute exists.

`hasattr(object, name)` checks if the object `object` has an attribute with a name stored in `name`:

```
s = "A string"
print(hasattr(s, "upper")) # True
print(s.upper())           # A STRING
```

Exercise:

Rewrite the `Greeter` example using `hasattr` instead of relying on `AttributeError`.

Code is available at `exercises/greeter_attributes.py`

Always having data

This code is inconvenient if we always expect to know our name, and uses a `try..except` block, which doesn't look very clean.

Let's require a name to be provided when a Greeter is created. For this, we need a special method called `__init__`.

This is analogous to constructor methods in other languages.

```python
class Greeter:
    def __init__(self, name):
        self.name = name

    def hello(self, target="World"):
        print(f"Hello, {target}!")
        print(f"My name is {self.name}.") # .name should always be defined
```

A meaningful constructor

```python
class Greeter:
    def __init__(self, name):
        self.name = name
```

When an object is created by calling `Greeter()`, it actually invokes this function for the class, which defaults to doing nothing.

By providing our own, we override this behavior, and require an extra argument:

```python
g = Greeter()
# TypeError: __init__() missing 1 required positional argument: 'name'

g = Greeter("Bob")
g.hello()
# Hello, World!
# My name is Bob.
```

If we wished, we could provide a default name as well.

Aside: "dunder" or "magic" methods

You will often see things named like `__something__` in Python.

`__init__` is a good example - whenever a new object is invoked, Python will run a method with this name.

Those are fixed names for objects interfacing with specific language functions.

Such names are surrounded by double underscores, hence *"dunder"*, or simply *"magic"*.

You should not create your own new names of this format, because a later version of Python can add more reserved magic names. Only add (override) those you know the function of.

Magic method example: get (re)presentable

If we try to print out our new object, it's still looks ugly.

```
print(g) # <__main__.Greeter object at 0x000001DB6B754760>
```

What is output when an object is evaluated as a string is determined by the `__str__` method. It can be used to make a readable representation.

Let's add a method:

```
    def __str__(self):
        return f"Greeter named {self.name}"
```

Now it will be used whenever we try to print the object:

```
g = Greeter("John")
print(f"g is a {g}") # Outputs "g is a Greeter named John"
```

Class variables

Instance properties such as `name` are unique to a specific instance of the Greeter class.

It's also possible to make a property that's shared by all instances of the class. Let's add a configurable default for `.greet()`:

```
class Greeter:
    default = "World"

    # ..rest of the code..
```

Now this is accessible both as `Greeter.default` and `g.default` on a specific instance `g`. This includes the `self` instance.

Let's try to use it in the `hello` method.

Parameter defaults gotchas

First naive attempt at using our new variable:

```
class Greeter:
    default = "World"
    # ..rest of the code..
    def hello(self, target=Greeter.default):
        print(f"Hello, {target}!")
        print(f"My name is {self.name}.")
```

This will raise an exception if the Greeter class is just being defined, because at the time of parsing it it's not yet defined, and function parameter defaults are evaluated at parse time.

Even if it was possible, it's not a good idea to assign a value that can change as a function default.

Parameter defaults gotchas

Another illustration of the same problem:

```python
test = "Something"
def f(x=test):
    print(x)
f() # Prints "Something"
test = "Other"
f() # Still prints "Something"
```

A more devious example:

```python
def f(l=[]):
    l.append("Test")
    return l

print(f()) # Outputs ["Test"]
print(f()) # Outputs ["Test", "Test"]
```

The common pattern for dealing with this problem is to assign `None` as default and replaces it within the function.

Class variables gotchas

```python
class Greeter:
    default = "World"
    # ..rest of the code..
    def hello(self, target=None):
        if target is None:
            target = self.default
        print(f"Hello, {target}!")
        print(f"My name is {self.name}.")
```

This should work.. Let's test it.

```python
g1 = Greeter("John")
g2 = Greeter("Jane")
g1.hello() # Outputs "Hello, World! My name is John."
g1.default = "class"
g1.hello() # Outputs "Hello, class! My name is John."
g2.hello() # Outputs "Hello, World! My name is Jane." ???
```

Shouldn't our change affect all instances?

Modifying class variables

`Greeter.default` is a class variable, but when we try to assign it on an instance, we actually create an instance variable that overrides it.

This behavior can actually be useful, especially for providing defaults, but that's not what we want here.

To properly modify an instance variable, we must access it through the class object, i.e. use `Greeter.default`:

```
g1 = Greeter("John")
g2 = Greeter("Jane")
g1.hello() # Outputs "Hello, World! My name is John."
Greeter.default = "class"
g1.hello() # Outputs "Hello, class! My name is John."
g2.hello() # Outputs "Hello, class! My name is Jane."
```

Usage counters

Exercise:

Add a method `.counts()` to the `Greeter` class that prints out how many times `.hello()` was called, both this specific instance and all instances together.

To keep track, you'll need to add new instance and class variables.

Base code is in `exercises/greeter_counts.py`

Example: inventory system

Let's make an inventory system for a computer game.

We have Item objects, which have three attributes: a name, a price, and a weight.

Item objects can be put in Containers (like a player inventory or a treasure chest), which have a name and a weight limit.

We already have an idea how to make an Item:

```python
class Item:
    def __init__(self, name, price, weight):
        self.name = name
        self.price = price
        self.weight = weight

    def __str__(self):
        return f"{self.name} ({self.price} gold) [{self.weight} kg]"


potion = Item("Potion of healing", 100, 0.2)
print(potion)
```

Specifying Containers

What do we want from our containers?

- We need to be able to add and remove objects from a container.
- A container can have multiple copies of an object in it.
- A container should not allow adding items over its weight limit.
- We don't care about the order in which we placed items in a container.
- We should be able to query the total weight, total value, and count of items in a container.
- We should be able to iterate through items in a container.

Container class

We need to keep track of multiple Item objects (and multiple copies) inside a Container. Let's use a dict to store counts and start writing:

```python
class Container:
    def __init__(self, name, weight_limit):
        self.name = name
        self.weight_limit = weight_limit
        self._counts = {}  # A dict that maps an Item to its count inside

    def count(self, item):
        return self._counts.get(item, 0)  # Returns 0 if item is not in dict

    def items_weight(self):
        return sum(item.weight * self.count(item) for item in self._counts)

    def items_price(self):
        return sum(item.price * self.count(item) for item in self._counts)

    def __str__(self):
        return f"{self.name} [{self.items_weight()}/{self.weight_limit} kg]"

inventory = Container("Player inventory", 50)
print(inventory)  # "Player inventory [0/50 kg]"
```

Implementation details

We could already start populating the Container by directly accessing `._counts`, but this has 2 problems:

1. This does not guarantee that weight limit is checked.
2. How items are stored in the class is an implementation detail that's subject to change.

So the class should provide its own methods for adding / removing items, and `._counts` should not directly be used.

```
potion = Item("Potion of healing", 100, 0.2)
inventory._counts[potion] = 1 # BAD: uses implementation details
inventory.add(potion)         # Good, but we still have to write this method!
```

Private attributes?

In many object-oriented languages, there is a concept of private properties and methods - they are only accessible by the methods of the class itself.

Python doesn't have such a concept: any attribute of a class is accessible.

By convention, attributes with names starting with an underscore (_) are considered private and *should not* be accessed externally.

```python
class Test:
    def __init__(self):
        self._private = "secret"
        self.__private = "top secret"

test = Test()
print(test._private)
print(test._Test__private) # Name is "mangled" but still accessible
```

Implementing the container

Let's add a method that adds an item.

```python
class Container:
    # ...
    def can_add(self, item):
        if not isinstance(item, Item):
            raise ValueError("Containers can only contain Items")
        return self.items_weight() + item.weight <= self.weight_limit

    def add(self, item):
        if self.can_add(item):
            old_count = self._counts.get(item, 0)
            self._counts[item] = old_count + 1
        else:
            raise RuntimeError(
                f"Can't add {item} to {self}: over weight limit"
            )
```

Note the use of `isinstance(object, class)` to check whether we're trying to add an `Item`.

Implementing the container

Exercise:

Implement a method `.remove(item)`

`.remove(item)` should remove the specified item from the container, and return it.

Raise an error (`KeyError`) if it's not in the container.

Base code is in `exercises/inventory1.py`.

Implementing an iterator

We want to be able to iterate over items in a Container:

```
for item in container:
  print(item)
```

How to implement this functionality? The standard Python answer is "magic methods".

Documentation: Special Method Names

Searching for "iterate", we find that we need a method `__iter__` that returns an iterator object.

Iterating with a generator

The most intuitive way to provide an iterator is to have a generator.

A generator can be thought of as a function that can "return" multiple times, pausing between calls. This is done with a `yield` keyword.

Iterating with a generator

The most intuitive way to provide an iterator is to have a generator.

A generator can be thought of as a function that can "return" multiple times, pausing between calls. This is done with a `yield` keyword.

Let's loop over the items in `_counts`, yielding the item as many times as the count of that object:

```
    def __iter__(self):
        for item in self._counts:
            for i in range(self.count(item)):
                yield item
```

Implementing other things

Exercise:

Implement `len(container)` and `x in container` operations.

This requires adding the `__len__` and `__contains__` methods.

Base code is in `exercises/inventory2.py`.

Homework exercise

Exercise:

Implement `inventory.loot(treasure)` method that takes items from the treasure container into the inventory container.

Note that the inventory may not be big enough to fit all items; your algorithm should try to optimize which items are taken to maximise total value.

Base code and a test case are provided in `exercises/inventoryH.py`

The example solutions provided are not optimal!

Class inheritance

One of the important tools in the object-oriented programming toolbox is *class inheritance:* ability to reuse code in one class by creating a subclass and expanding / changing the base class functionality.

Let's make a subclass of Item, Weapon. It adds an extra attribute, `dps` (damage per second).

Note that any Weapon will be both a Weapon and an Item.

We can reuse methods and properties coming from the base class; in case we override a method in the subclass, we can use `super()` to access the base class method.

Class inheritance

```python
class Weapon(Item):
    def __init__(self, name, price, weight, dps):
        super().__init__(name, price, weight)
        # Equivalent code: Item.__init__(self, name, price, weight)
        self.dps = dps

    def __str__(self):
        return super().__str__() + f" {{{self.dps} DPS}}"

sword = Weapon("Broadsword", 50, 5, 10)
print(sword) # Outputs "Broadsword (50 gold) [5 kg] {10 DPS}"

inventory = Container("Player inventory", 50)
inventory.add(sword) # No error: a Weapon is an instance of Item
```

Exercise:

Implement a container method `.best_weapon()` that returns the weapon with highest DPS, or `None` if there are no weapons in the container.

Base code is in `exercises/inventory3.py`.

Multiple inheritance

A single class can inherit from multiple base classes.

In this case, the order of classes matters for resolving which method to use:

```python
class A:
    def x(self):
        print("x from A")

    def y(self):
        print("y from A")

class B:
    def x(self):
        print("x from B")

class C(B, A):
    pass

class D(A, B):
    pass

C().x() # Prints "x from B"
C().y() # Prints "y from A"
D().x() # Prints "x from A"
```

Aside: getters and setters

Sometimes you need an object property that is dynamically computed or that needs processing when assigned.

A *getter* is a function returning a value as a property, while a *setter* is a function that receives the value when a property is written to. This can be achieved with a built-in `property`:

```python
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"

p = Person("John", "Doe")
print(p.full_name)
p.full_name = "Test" # AttributeError: can't set attribute
```

This uses new syntax `@something`, which will be covered later.

Aside: getters and setters

Example with a setter:

```python
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return f"{self.first_name} {self.last_name}"

    @full_name.setter
    def full_name(self, name):
        components = name.split(" ")
        if len(components) == 2:
            self.first_name = components[0]
            self.last_name = components[1]
        else:
            raise ValueError("Can't determine first/last name")

p = Person("John", "Doe")
p.full_name = "Jane Doe"
p.full_name = "Superman" # ValueError
```

Multiple inheritance

Exercise:

Implement a class `PortableContainer` that is both an Item and a Container.

Replace the `weight` and `price` properties with getters that calculate total weight and value of the container and its contents.

The constructor is already provided for you.

Base code and test case are in `inventory4.py`

Functional programming

What is functional programming?

Functional programming is a style of programming that emphasizes use of "pure" functions:

- Functions that do not depend on external or internal state (only on input)
- Functions that don't change anything in the external state (no "side effects")

The advantages of such functions is ease of reasoning about the program - there are fewer surprises.

Besides function purity, another big part of functional programming is the ability to treat functions as values: so that a function can itself be a parameter to another function.

Impurity example

```python
def add_zero(l):
    l.append(0)

def increment_all(l):
    for i in range(len(l)):
        l[i] += 1
```

Those functions are impure, because they change the list `l`.

Their result depends on order of execution:

```python
l = []
add_zero(l)
increment_all(l)
print(l) # Outputs [1]

l = []
increment_all(l)
add_zero(l)
print(l) # Outputs [0]
```

Impurity example

```python
increment = 0
def increasing_increment(num):
  global increment
  increment += 1
  return num += increment

increasing_increment(0) # Returns 1
increasing_increment(0) # Returns 2
```

This function depends on global state. It can be very hard to debug.

We have examples of this behavior in our `Greeter` class.

Functional programming and immutability

If we want functional programming purity, we can't have operations that change something "in-place". We must return a copy of the data that has the required modifications, leaving the original data intact.

```python
def increment_all(l):
    new_l = l.copy()
    for i in range(len(new_l)):
        new_l[i] += 1
    return new_l

l = [0, 1]
print(increment_all(l)) # Outputs [1, 2]
print(l)                # Outputs [0, 1]
```

The best way to ensure immutability is using data structures that are themselves immutable. It is often not easy to achieve in Python, however.

Aside: shallow and deep copies

As mentioned before, objects in Python can contain references to other objects that are, themselves, mutable.

Trying to copy objects can also run into this problem:

```python
def append_zero_all(l):
    new_l = l.copy()
    for sublist in new_l:
        sublist.append(0)
    return new_l

l = [[1], []]
new_l = append_zero_all(l)
print(new_l) # Outputs [[1, 0], [0]]
print(l)     # Outputs [[1, 0], [0]] ???
```

`l.copy()` performs a *shallow* copy of a list; a new list is created with references to the same objects as the original. For primitive values it's not a problem: they are immutable. But for mutable objects, it is.

Aside: shallow and deep copies

To create a "proper" copy of an object with references to potentially mutable objects, you need a recursive "deep" copy. Thankfully, the standard library provides tools for this:

```python
from copy import deepcopy
def append_zero_all(l):
    new_l = deepcopy(l)
    for sublist in new_l:
        sublist.append(0)
    return(new_l)

l = [[1], []]
new_l = append_zero_all(l)
print(new_l) # Outputs [[1, 0], [0]]
print(l)    # Outputs [[1], []]
```

Exercise:

Modify the `Container.add()` method to be pure: instead of changing the instance, return a modified new instance.

Base code is in `exercises/inventory5.py`

Higher-order functions

As mentioned, Python treats functions as values; so they can be passed to other functions.

Functions that accept functions as parameters are called *higher-order functions*.

```python
def run_twice(f):
    f()
    f()

def test():
    print("Test")

run_twice(test) # Outputs "Test" twice
```

Functional programming tools

Python provides some standard primitives for functional programming: some built-in, some in the `functools` module.

```python
from functools import reduce
l = ["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]

def f(acc, val):
    return f"{acc} {val}"

def g(word):
    return word.capitalize()

print(reduce(f, map(g, l))
# Outputs "The Quick Brown Fox Jumps Over The Lazy Dog"
```

Generators and lazy evaluation

If you use `map()`, you may notice that it returns something other than `list`

```
squares = map(lambda x: x**2, [1, 2, 3, 4])
squares[2] # TypeError: 'map' object is not subscriptable
```

This looks inconvenient, but `map` actually returns a *generator:* an iterable which is evaluated one value at a time when iterating. This is an example of lazy evaluation.

As a result, it can actually work on infinite iterables:

```
from itertools import count
all_numbers = count() # Iterable returning 0, 1, 2, 3, ...
all_squares = map(lambda x: x**2, count())
for num, squared in enumerate(all_squares):
    print(f"{num} squared is {squared}")
    if num == 10:
        break
```

Decorators

Higher-order functions, again

Functions can return functions as well:

```python
def run_twice(f):
    def twice_f():
        f()
        f()
    return twice_f

def test():
    print("Test")

twice_test = run_twice(test)
twice_test() # Outputs "Test" twice
```

In this, `f` is called a *wrapped* function and `twice_f` is called a *wrapper*.

This is a common enough pattern to have special syntax: decorators.

Decorators

```
def run_twice(f):
    def twice_f():
        f()
        f()
    return twice_f

@run_twice
def test():
    print("Test")

test() # Outputs "Test" twice
```

This is equivalent to running `test = run_twice(test)`

Dealing with arguments and return values

You can pass through arguments to the wrapped function and return a value out of the wrapper.

```python
def run_twice(f):
    def twice_f(*args, **kwargs):
        f(*args, **kwargs)
        return f(*args, **kwargs)
    return twice_f

@run_twice
def hello(target):
    print(f"Hello, {target}!")
    return 42

print(hello("World")) # Outputs "Hello, World!" 2 times, then 42
```

Decorator parameters

You can have decorator with parameters if you make a function that returns a decorator:

```
def run_multiple(num):
    def decorator_multiple(f):
        def wrapper_multiple(*args, **kwargs):
            for i in range(num):
                result = f(*args, **kwargs)
            return result
        return wrapper_multiple
    return decorator_multiple

@run_multiple(3)
def hello(target):
    print(f"Hello, {target}!")
    return 42

print(hello("World")) # Outputs "Hello, World!" 3 times, then 42
```

A practical example: debug information

Let's write a decorator that helps with tracing when a function is called.

```
def debug(f):
    def debug_wrapper(*args, **kwargs):
        print(f"{f.__name__} called with arguments:", *args, **kwargs)
        return f(*args, **kwargs)
    return debug_wrapper

@debug
def hello(target):
    print(f"Hello, {target}!")
    return 42

print(hello("World"))
# hello called with arguments: World
# Hello, World!
# 42
```

Stacking decorators

We can apply two decorators at once:

```python
@run_twice
@debug
def hello(target):
    print(f"Hello, {target}!")

hello("World")
# hello called with arguments: World
# Hello, World!
# hello called with arguments: World
# Hello, World!
```

The function is wrapped in the inverse order of decorators.

Decorators and function identity

Let's try the other way around:

```
@debug
@run_twice
def hello(target):
    print(f"Hello, {target}!")

hello("World")
# twice_f called with arguments: World
# Hello, World!
# Hello, World!
```

Notice that the name of the function after being wrapped by `@run_twice` changes. Sometimes it's not desirable; `functools` has a solution.

Decorators and function identity

```python
from functools import wraps

def run_twice(f):
    @wraps(f)
    def twice_f(*args, **kwargs):
        f(*args, **kwargs)
        return f(*args, **kwargs)
    return twice_f

@debug
@run_twice
def hello(target):
    print(f"Hello, {target}!")

hello("World")
# hello called with arguments: World
# Hello, World!
# Hello, World!
```

Tracing recursive calls

A classic example of recursive computation are the Fibonnacci numbers.

```python
@debug
def fib(n):
    if n in [0, 1]:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

This naive implementation results in a lot of overhead, because the recursive computation branches, multiple sub-calls computing the same value, which we can observe with `@debug` wrapper

Tracing recursive calls

```
print(fib(4))
# fib called with arguments: 4
# fib called with arguments: 2
# fib called with arguments: 0
# fib called with arguments: 1
# fib called with arguments: 3
# fib called with arguments: 1
# fib called with arguments: 2
# fib called with arguments: 0
# fib called with arguments: 1
# 5
```

Our `fib(n)` function is pure: its return value only depends on the argument `n`.

This can be used to optimize this code by caching intermediate results, which is called *memoization*.

Memoization

Exercise:

Write a decorator `@memoize` that stores a dictionary of already computed results of its single-argument function, and uses it as cache.

Base code is in `examples\fib.py`

Note:

Similar functionality is provided by `@functools.lru_cache` decorator from the standard library.

Modules and packages

What are modules and packages?

Modules are a mechanism for sharing code across multiple Python source files. A module is a single `.py` file.

It helps logically organize longer code, as well as allow using your code as a library for other code.

Packages are a way to organize multiple modules together in a tree-like structure of submodules.

Python libraries that are built-in or installable are packages that you `import` into your code.

Importing code

```python
import foo
# Everything in the global context of foo.py is now available under foo.*

from baz import quux, quuz
# Global objects quux and quuz from baz.py are added in the current context

from monty import spam as ham
# Global object spam from monty.py is now available under the name ham
import foo as bar
```

Besides those 3, there's a another mechanism called *star-import:*

```python
from junkyard import *
# Import into current context
# If the module defines a list __all__:
#   Import all global objects that are in that list
# Else:
#   Import all global objects that don't start with "_" ("private")
```

Because it modifies the global namespace in an unpredictable way, the use of star-imports is discouraged.

Module example

Suppose we have the following code in `spam.py`:

```python
def spamalot():
    print("SPAM " * 100)

print("I will now spam a lot!")
spamalot()
```

We find this function very useful, and want to import it from another script, `imports.py`, in the same folder.

```python
import spam
from spam import spamalot

print("Trying imports...")
spamalot()
spam.spamalot()
```

Exercise:

Those files are under `exercises`; try running `imports.py`

What happens on import?

If we run `imports.py`, we see the following:

```
I will now spam a lot!
SPAM SPAM SPAM SPAM SPAM ...
Trying imports...
SPAM SPAM SPAM SPAM SPAM ...
SPAM SPAM SPAM SPAM SPAM ...
```

As expected, both our calls are executed, but why it is executed once more before?

To be able to import code, Python needs to run the module first, populating its global context. This can be used for initialization, but in this case it also includes our test code.

Note that it happens only once: Python caches the result of running the module for future imports during the current execution.

What else happens on import?

You may notice something else: a folder called `__pycache__` appeared.

When a module is imported, it is first parsed into an optimized structure for execution. Python saves this optimized version to skip this step next time the module has to be imported.

If you change a module, Python will automatically detect that the cache is out of date and will rebuild it.

How do we avoid too much spam?

Our test code only makes sense if we're running `spam.py` directly. How to detect that?

At runtime, the magic variable `__name__` contains the name of the current module.

However, if a module is executed directly, `__name__` will be set to a special name "`__main__`".

```
if __name__ == "__main__":
  print("We're executed directly")
else:
  print(f"We're loaded as a module {__name__}")
```

Exercise:

Modify `spam.py` to only execute the test code if it is run directly.

Where does Python search for modules?

The full list of locations is available as `sys.path` from the standard module `sys`.

The list is populated as follows:

1. The directory containing the input script (or the current directory when no file is specified).
2. `PYTHONPATH` environment variable (a list of directory names, with the same syntax as the shell variable `PATH`).
3. The installation-dependent default.

It's possible to modify `sys.path` at runtime.

Packages

Packages are modules organized in subfolders.

When published, a package contains additional metadata describing the package, listing its dependencies and install instructions.

As an example, there's a package `mypackage` under `exercises`, which consists of 4 files, and a script `imports2.py` that uses it.

Exercise:

Examine the files. Try running `imports2.py`.

Can you guess how `__init__.py` works?

Python environments

What's a Python environment?

A Python *environment* is, roughly speaking, a specific Python version plus a collection of extra packages.

When creating a new Python environment for a specific project, they are usually called *virtual* environments.

What problem do environments solve?

Imagine there exists an AwesomeLibrary. It's awesome, so it's quite popular.

One of the projects you're working on requires it. However, it requires AwesomeLibrary 1.x.x, since it hasn't been updated yet to changes in AwesomeLibrary 2.x.x.

On the other hand, you're also working on another project, that expects AwesomeLibrary 2.x.x. You'll note that the `import` mechanism doesn't have anything to do with versions, so how can you keep working on both?

The solution is virtual environments.

What's needed for a virtual environment?

Working with virtual environments require 2 pieces of software:

- A package manager to install packages in the environment. The standard approach here would be `pip`
- A virtual environment manager that sets up the environment. Historically the most used is `virtualenv`, while with Python 3.6+ there is a standard module `venv`.

Note:

Despite it being part of the standard library, on Linux systems an extra installation is sometimes needed for `venv`

Virtual environment workflow

1. Creating a virtual environment
2. Entering ("activating") virtual environment for the current shell
3. Installing dependencies
4. Running code while the virtual environment is active

Virtual environment example

There's a script `exercises/weather.py` that requires non-standard packages.

Let's make a virtual environment to run it.

```
python -m venv training-env
```

This creates a new folder `training-env` containing everything needed for a virtual environment.

To activate it, we need to execute a script; depends on the OS:

```
source training-env/bin/activate
```

```
training-env\Scripts\activate
```

Virtual environment example

Once the environment is activated for the current shell, running `python` and `pip` will use the environment's Python and packages.

Exercise:

Try running `exercises/weather.py` with the environment.

Install missing packages until it runs:

```
pip install [package_name]
```

To avoid guesswork next time we need to create the environment, we can save the requirements:

```
pip freeze > requirements.txt
# Later
pip install -r requirements.txt
```

Tricky parts of Python package management

Some packages rely on non-python code and libraries.

- Those libraries may need to be installed separately.
- Plugin code may need to be compiled, requiring installing build tools and development versions of libraries.

`pip` supports a binary format called "wheels", but it doesn't always help with the above problems.

What about (Ana)conda?

`conda` is a package / virtual environment manager that is part of the Anaconda distribution.

Among other things, it can manage Python packages.

Differences:

- Conda environments are saved in the user profile by default
- Conda has more support for binary packages (e.g. installing libraries together with packages)
- Conda doesn't support everything `pip` does, but can interoperate with it.

End of material

Extras?

Thank you!

Questions?