Effective Programming Practices for Economists

14. Object-oriented programming

Hans-Martin von Gaudecker

Department of Economics, Universität Bonn

universitätbonn

Revisiting abstractions
Reminder of the example

Reconsider the prospect theory example:

$$u(z,\gamma,\lambda) = \begin{cases} z^{1-\gamma} & \text{for } z \geq 0 \\ -\lambda \cdot (-z)^{1+\gamma} & \text{for } z < 0 \wedge \gamma < 0 \\ -\lambda \cdot (-z)^{1-\gamma} & \text{for } z < 0 \wedge \gamma \geq 0 \end{cases}$$

where $z \in \mathbb{R}_+$, $|\gamma| < 1$, and $\lambda \geq 1$.

Revisiting abstractions
Cases to consider

z	γ	λ
100	0.5	.
100	0.0	.
-100	0.5	2.0
-100	0.0	2.0

Revisiting abstractions
No abstraction at all

```
>>> # Show result for case 1 with z=100 and gamma = 1/2
>>> print(100.0 ** (1.0 - 0.5))
10.0

>>> # Show result for case 2 with z=100 and gamma = 0
>>> print(100.0 ** (1.0 - 0.0))
100.0

>>> # Show result for case 3 with z=-100, gamma = 1/2, and lambda = 2
>>> print((-2) * (-(-100.0)) ** (1 - 0.5))
-20.0

>>> # Show result for case 4 with z=-100, gamma = 0, and lambda = 2
>>> print((-2) * (-(-100.0)) ** (1 - 0.0))
-200.0
```

Revisiting abstractions
Avoiding magic numbers

```
>>> # Show result for case 1 with z=100 and gamma = 1/2
>>> z = 100
>>> gamma = 0.5
>>> print(z ** (1 - gamma))
10.0
>>> # Show result for case 2 with z=100 and gamma = 0
>>> gamma = 0.0
>>> print(z ** (1 - gamma))
100.0
>>> # Show result for case 3 with z=-100, gamma = 1/2, and lambda = 2
>>> z = -100
>>> gamma = 0.5
>>> lambda_ = 2.0
>>> print((-lambda_) * (-z) ** (1 - gamma))
-20.0
>>> # Show result for case 4 with z=-100, gamma = 0, and lambda = 2
>>> gamma = 0.0
>>> print((-lambda_) * (-z) ** (1 - gamma))
-200.0
```

Revisiting abstractions
Adding functions

```
>>> def utility_prospect_theory(z, gamma, lambda_=None):
...     if z >= 0:
...         u = z ** (1 - gamma)
...     elif z < 0:
...         if gamma <= 0:
...             u = -lambda_ * (-z) ** (1 + gamma)
...         elif gamma > 0:
...             u = -lambda_ * (-z) ** (1 - gamma)
...     return u

>>> # Show result for case 1 with z=100 and gamma = 1/2
>>> z = 100
>>> gamma = 0.5
>>> print(utility_prospect_theory(z, gamma))
10.0
```

Revisiting abstractions
Adding complex data structures: lists

```
>>> # Put all parameters in a nested list.
>>> parameters = [
...     [100.0, 0.5, None],
...     [100.0, 0.0, None],
...     [-100.0, 0.5, 2.0],
...     [-100.0, 0.0, 2.0]
... ]

>>> for p in parameters:
...     print(utility_prospect_theory(p[0], p[1], p[2]))

10.0
100.0
-20.0
-200.0
```

Revisiting abstractions
Adding complex data structures: dictionaries

```
>>> # Put all parameters in a list of dictionaries.
>>> parameters = [
...     {'z': 100.0, 'gamma': 0.5, 'lambda': None},
...     {'z': 100.0, 'gamma': 0.0, 'lambda': None},
...     {'z': -100.0, 'gamma': 0.5, 'lambda': 2.0},
...     {'z': -100.0, 'gamma': 0.0, 'lambda': 2.0}
... ]

>>> for p in parameters:
...     print(utility_prospect_theory(p['z'], p['gamma'], p['lambda']))

10.0
100.0
-20.0
-200.0
```

A fresh look at the structure of our example

- Agent, described by:
  - Utility function
  - Parameters of this function
- Goal: Calculate her utility evaluation of a monetary value

A fresh look at the structure of our example

- So what if we had an object (“the agent”) that . . .
  - . . . would have utility function parameters as attributes . . .
  - . . . and a method that returns her utility evaluation of a number that is passed in?
- Classes give us exactly this functionality
  - *A class packs a set of data together with functions operating on that data.* (Langtangen, 2009)
  - Highly recommended for introductions to classes in a scientific context

Calculating the utility of agents

```
donald = AgentProspectTheory(gamma=0.5, lambda_=2.0)
daisy = AgentProspectTheory(gamma=0.0, lambda_=2.0)

z_values = [100, -100]
for z in z_values:
    print(donald.utility(z))
    print(daisy.utility(z))
```

```
10.0
100.0
-20.0
-200.0
```

This ain't new . . .

- We have used this pattern all the time
  - Create a string and use its `replace` method
  - Create a NumPy array and get its transpose:

```
>>> import numpy as np
>>> a = np.array([[1, 2], [3, 4]])
>>> a.T
array([[1, 3],
       [2, 4]])
```

- The only innovation is that we defined a custom class of objects called `AgentProspectTheory`

The class / object distinction

- A class in Python (Matlab, Ox, Java, C++, ...) has the same meaning as in mathematics
  - A family of things sharing the same characteristics
  - Example: All quadratic functions:

$$f(x) = a_0 \cdot x^0 + a_1 \cdot x^1 + a_2 \cdot x^2$$

- Must specify values for a_0, a_1, a_2 to calculate the function value of a particular quadratic function

The class / object distinction

- OOP-speak:
  - $f_0(x) = 2 + 6x + 7x^2$ is an **instance** of the class of quadratic functions
- Again, we have used this all the time
  - The object `gamma` was an instance of the `float` class
  - The object `parameters` was an instance of the `list` class
  - And so on

What good does it do?

- Variable names allowed us to re-use values in different places
- Functions and loops allowed us to re-do similar calculations repeatedly
- Lists and dictionaries allowed us to bundle (related) values
- Custom classes can bundle (related) values **and** functions

What good does it do?

- (Once our tests are in place,) we don't have to worry anymore about . . .
  - . . . how agents calculate their utility or . . .
  - . . . how to pass preferences into a utility function
- We just call `daisy.utility(100)`

What good does it do?

- This programming pattern is called **encapsulation**
  - Anything we need to describe an agent is bundled in the appropriate class instance itself
- Really helpful when other things are complicated as well
  - In this example: The choice problem
  - Natural to describe vertical and horizontal relationships among custom objects

```python
class AgentProspectTheory:
    """Define an agent described by a prospect theory-type utility
    function and its parameters.

    """

    def __init__(self, gamma, lambda_):
        """Special function called upon creation of a class instance.

        Set the utility function parameters.

        """

        self.gamma = gamma
        self.lambda_ = lambda_

    def utility(self, z):
        """Return the agent's utility evaluation of a monetary value z."""
        if z >= 0:
            u = z ** (1 - self.gamma)
        elif z < 0:
            if self.gamma <= 0:
                u = -self.lambda_ * (-z) ** (1 + self.gamma)
            elif self.gamma > 0:
                u = -self.lambda_ * (-z) ** (1 - self.gamma)
        return u
```

Dissecting the `AgentProspectTheory` class

```
class AgentProspectTheory:
    def __init__(self, gamma, lambda_):
        self.gamma = gamma
        self.lambda_ = lambda_
```

- Create an instance of the `AgentProspectTheory` class by writing:

```
AgentProspectTheory(gam, lam)
```

- This triggers a call to the class' `__init__` method, which is also known as a **constructor**

Dissecting the `AgentProspectTheory` class

```
class AgentProspectTheory:
    def __init__(self, gamma, lambda_):
        self.gamma = gamma
        self.lambda_ = lambda_
```

- Since it takes exactly two arguments, we get a familiar error if we supply only one:

```
uncle_scrooge = AgentProspectTheory(lambda_=100.0)
```

```
Traceback (most recent call last):
  File "agent.py", line 41, in <module>
    uncle_scrooge = AgentProspectTheory(lambda_=100.0)
TypeError: __init__() takes exactly 3 arguments (2 given)
```

The main stumbling block: Explaining classes' `self` parameter

- Python automatically passes a reference to the **object** itself as the first argument to **any** method of the class
- By (strict) convention, this is called `self`
- Why?
  - There are no global variables inside a class
    (the usual rules for module-level globals apply)
  - Inside class methods, the `self` reference to the object itself is needed in order to access its other methods / variables

The main stumbling block: Explaining classes' `self` parameter

- This leads to the following rules (Langtangen, 2009):
  - `self` must be the first argument of any method **definition**
  - To access another class method or to access / set a class attribute, inside class methods, you must prefix with `self`
  - In **calls** to class methods, `self` is dropped as an argument. You call the methods only with the remaining arguments
  - Only method definitions can appear in the indented block under the class headline. You thus need to do all assignments etc. inside methods

Dissecting the `AgentProspectTheory` class

```
class AgentProspectTheory:
    def utility(self, z):
        if z >= 0:
            u = z ** (1 - self.gamma)
        elif z < 0:
            if self.gamma <= 0:
                u = -self.lambda_ * (-z) ** (1 + self.gamma)
            elif self.gamma > 0:
                u = -self.lambda_ * (-z) ** (1 - self.gamma)
        return u
```

- Create class methods like functions in a module
- Use `self` as a method's first parameter, and access class variables using `self.variable_name`

Dissecting the `AgentProspectTheory` class

```
class AgentProspectTheory:
    def utility(self, z):
        if z >= 0:
            u = z ** (1 - self.gamma)
        elif z < 0:
            if self.gamma <= 0:
                u = -self.lambda_ * (-z) ** (1 + self.gamma)
            elif self.gamma > 0:
                u = -self.lambda_ * (-z) ** (1 - self.gamma)
        return u
```

- Call an instance method with the usual dot-notation and without the `self` argument, i.e.

```
daisy.utility(100)
```

Class special and private methods

- Python provides some special methods you can attach to classes, which are invoked upon specific events
- E.g. the `__init__` method is called when an instance of the class is created
- All **special methods** have two leading and trailing underscores

Class special and private methods

- Definition **private methods and variables**:
  - Those that should not be accessed directly from calling code
- By convention, private methods and variables start with a single underscore
  - Thus, special methods are private as well
- You could call, e.g. `daisy.__init__(2.5, 1.0)` after creating `daisy`, but it is good practice to never do this
  - I.e. do not abuse special and private methods

Some important special methods: `__call__`

The `__call__` method is invoked when a class instance is being called like a function

```
class AgentUtilityProspectTheory:
    """[Same as before, but an instance call returns utility]"""

    def __init__(self, gamma, lambda_):
        self.gamma = gamma
        self.lambda_ = lambda_

    def _utility(self, z):
        """[Same as before, but hide it from the outside world]"""

    def __call__(self, z):
        return self._utility(z)


dewey = AgentUtilityProspectTheory(gamma=0.5, lambda_=2.0)
print(dewey(100))
print(dewey.__call__(100))  # Don't do this!!!
print(dewey._utility(100))  # Don't do this!!!
```

Some important special methods: `__str__`

The `__str__` method is invoked when a string representation of a class instance is requested[†], e.g. in `print` statements, the `str()` conversion function, or the `.format()` method

```
# The AgentProspectTheory class does not have a __str__ method.
daisy = AgentProspectTheory(gamma=0.0, lambda_=2.0)
print(daisy)
```

```
<__main__.AgentProspectTheory object at 0x101071390>
```

[†]For completeness: There also is a `__repr__` method, which is invoked under slightly different circumstances. You usually won't care, else check the Python documentation

Some important special methods: `__str__`

```
class AgentUtilityProspectTheory:

    def __str__(self):
        return """
Return the utility evaluation of a monetary value of an
agent with power utility and loss aversion, coefficients:
gamma = {g}
lambda_ = {l}""".format(g=self.gamma, l=self.lambda_)


# This string representation is meaningful for humans.
print(dewey)
```

```
Return the utility evaluation of a monetary value of an
agent with power utility and loss aversion, coefficients:
gamma = 0.5
lambda_ = 2.0
```

Inspecting an object's attributes

The built-in function `dir` returns a list of an object's attributes:

```
# Get a list of all instance attributes.
print(dir(daisy))
```

```
['__class__', '__delattr__', '__dict__', '__doc__',
'__format__',  '__getattribute__', '__hash__', '__init__',
'__module__', '__new__',  '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__',  '__str__',
'__subclasshook__', '__weakref__',
'gamma', 'lambda_', 'utility']
```

Most of these you'll never need

A look at some built-in attributes

For an instance of the `AgentUtilityProspectTheory` class:

```
print(dewey.__module__)  # The module where the class was defined.
print(dewey.__class__)   # The instance's class.
print(dewey.__dict__)    # The instance's variables dictionary.
print(dewey.__doc__)     # The docstring of the class.
```

```
__main__

<class '__main__.AgentUtilityProspectTheory'>

{'lambda_': 2.0, 'gamma': 0.5}

[Same as before, but an instance call returns utility]
```

Some important special methods:

`__getitem__`

Make class instances behave like (read-only) dictionaries:

```
class AgentUtilityProspectTheory:

    def __getitem__(self, key):
        return self.__dict__[key]


# AgentUtilityProspectTheory instances now behave like a (read-only)
# dictionary without the .keys() methods etc. (easy to define as well).
print(dewey['gamma'])
print(dewey.__dict__['gamma'])  # Don't do this!!!
```

```
0.5
0.5
```

Some important special methods: Wrapping up

- A corresponding `__setitem__` special method also exists
- Together with `__len__`, you can easily emulate list behaviour and much more
- You can find a full list of special methods at:

  `http://docs.python.org/3/reference/datamodel.html#special-method-names`

Special methods, operator overloading, and polymorphism

- Polymorphism means “having more than one form”
- In programming, it refers to an expression involving a variable can do different things depending on the type of the object to which the variable refers (Campbell et al., 2009)
- For example, the + operator means different things for different objects

```
>>> left + right # 'left' + 'right'
'leftright'
>>> left + right # 3 + 8
11
```

Special methods, operator overloading, and polymorphism

- This is called **operator overloading**
- E.g., for your own classes, you could define an `__add__` special method
  - Will be called when an instance stands to the left of a `+`

Polymorphism and interchangeability of objects

- Polymorphism means that you can substitute one object for another, as long as:
  - Both provide the relevant method
  - Those methods have the same **interface**

Example of polymorphism

```
>>> class NumberWithoutTypeCheck:
...
...     def __init__(self, value):
...         self.value = value
...
...     def __add__(self, x):
...         return self.value + x
...

>>> x = NumberWithoutTypeCheck(5.0)
>>> x + 7
12.0

>>> 7 + x
TypeError: unsupported operand type(s) for +: 'int' and 'instance'

>>> y = NumberWithoutTypeCheck('abc')
>>> y + 7
TypeError: cannot concatenate 'str' and 'int' objects
```

Polymorphism and code re-use

- Polymorphism and proper encapsulation mean that it is **VERY** simple to change your code
- If you want to replace a set of objects:
  - Define a new class with the same (subset of the) interface
  - Use instances of the new class instead of old objects
- E.g., to create something that can be used in place of a file, just create a class with methods `read`, `readlines`, etc.
- For this, it helps to only ever interact with classes via methods rather than attributes
  - No need to care how data is represented internally

Inheritance

- Suppose we wanted to calculate certainty equivalents for different utility functions u:

$$\mu(\tilde{z}, u) = u^{-1}\left(\mathbb{E}\left[u(\tilde{z})\right]\right)$$

- We could re-write the calculation of the expectation for every u
  - **DRY!!!**

Inheritance

- With classes, it is easy abstract from functions . . .
- The mechanics are the same regardless of the specific functions u and u^{-1}
  - Apply u to (monetary) values.
  - Calculate the expectation
  - Apply u^{-1} to this expectation
- No need to pass preference parameters into functions since they are attributes of the object itself

Inheritance

- Apply this idea by . . .
  - . . . defining a superclass `AgentRiskyProspects` with methods to calculate certainty equivalents
  - . . . letting classes with utility functions **inherit** those methods from `AgentRiskyProspects`

Inheritance

```
class AgentRiskyProspects:

    """Superclass for an agent who makes utility evaluations of risky
    prospects. Implement everything that is not specifically related
    to a particular utility function.

    """

    def certainty_equivalent(self, prospects, probabilities):
        """Return the agent's certainty equivalent, given a vector
        of prospects and corresponding probabilities.

        """
        exp_u = self.expected_utility(prospects, probabilities)
        return self.utility_inverse(exp_u)

    def expected_utility(self, prospects, probabilities):
        """Return the agent's expected utility, given a vector
        of prospects and corresponding probabilities.

        """
        return np.inner(self.utility(prospects), probabilities)

    def __getitem__(self, key):
        return self.__dict__[key]
```

Inheritance in action

- Note that the `utility` and `utility_inverse` methods are **not** defined in `AgentRiskyProspects` objects
  - It is pointless to create an instance of the class
- Define the inheritance relationship in the class definition
  - Specify the parent class like an argument in a function call
- Here, `AgentKinked` inherits all of `AgentRiskyProspects`' attributes

```
class AgentKinked(AgentRiskyProspects):
```

Inheritance in action

```
class AgentKinked(AgentRiskyProspects):

    """An agent described by a kinked utility function,
    which only has a loss aversion coefficient lambda_."""

    def __init__(self, lambda_):
        assert lambda_ > 0, 'Loss aversion must be positive.'
        self.lambda_ = lambda_

    def utility(self, z):

    def utility_inverse(self, x):


huey = AgentKinked(lambda_=2.0)
print(huey['lambda_'])


2.0
```

Inheritance in action

```
prospects = np.array([-20, 40])
probabilities = np.array([.5, .5])
print(huey.certainty_equivalent(prospects, probabilities))
```

```
0.0
```

- Remember the structure:
  - `certainty_equivalent` and `expected_utility` were both defined in the superclass
  - `utility` and `utility_inverse`, on which the former operate, were both defined in the subclass

Inheritance in action

- Using inheritance properly is an extremely powerful device for minimising repetition
  - When you need similar but not identical objects, group the shared characteristics in a class
  - Then let the actual objects inherit from that class

```
>>> help(huey)

class AgentKinked(AgentRiskyProspects)
 |  An agent described by a kinked utility function,
 |  which only has a loss aversion coefficient lambda_.
 |
 |  Method resolution order:
 |      AgentKinked
 |      AgentRiskyProspects
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(self, lambda_)
 |
 |  __str__(self)
 |
 |  utility(self, z)
 |      Return piece-wise linear utility:
 |
 |      u = z for z >= 0
 |      u = z * self.lambda_ for z < 0
 |
 |      Designed to handle scalars and NumPy arrays.
 |
 |  utility_inverse(self, x)

[...]

 |  ----------------------------------------------------------------------
 |  Methods inherited from AgentRiskyProspects:
 |
 |  __getitem__(self, key)
 |
 |  certainty_equivalent(self, prospects, probabilities)
 |      Return the agent's certainty equivalent, given a vector
 |      of prospects and corresponding probabilities.

[...]
```

Method resolution order

- If two classes in a hierarchical relationship methods with the same name, the one from the inheriting code is used
- This means you can override methods from a higher level for special cases
- Imagine a class for the exponential family of distributions
  - You have a method calculating the expected value from the moment generating function
  - When the normal distribution inherits its characteristics from the exponential family, you override the method (just read the expected value off its parameters)

Method resolution order: Example

Easy to forget or mistype the functions `utility` and `utility_inverse` in the subclass.

Better define **placeholders** in the superclass, which do nothing but to raise meaningful errors:

```
class AgentRiskyProspects:

    def utility(self, z):
        """Placeholder, should be overridden by subclass."""
        raise AttributeError(IMPLEMENT_U_MSG)

    def utility_inverse(self, x):
        """Placeholder, should be overridden by subclass."""
        raise AttributeError(IMPLEMENT_U_MSG)
```

Method resolution order: Example

The placeholders will only ever get called if the subclass did not correctly define `utility` and `utility_inverse`:

```
huey = AgentKinked(lambda_=2.0)
print(huey.utility(20))

louie = AgentRiskyProspects()
print(louie.utility(20))
```

```
20.0

Traceback (most recent call last):
  File "inheritance_placeholder.py", line 134, in <module>
    print(louie.utility(20))
  File "inheritance_placeholder.py", line 31, in u
    raise AttributeError(IMPLEMENT_U_MSG)
AttributeError: You have to define methods utility and
utility_inverse in the subclass which inherits from
AgentRiskyProspects. Both should take one argument
(monetary values / a scalar expected utility evaluation).
```

BHT example revisited

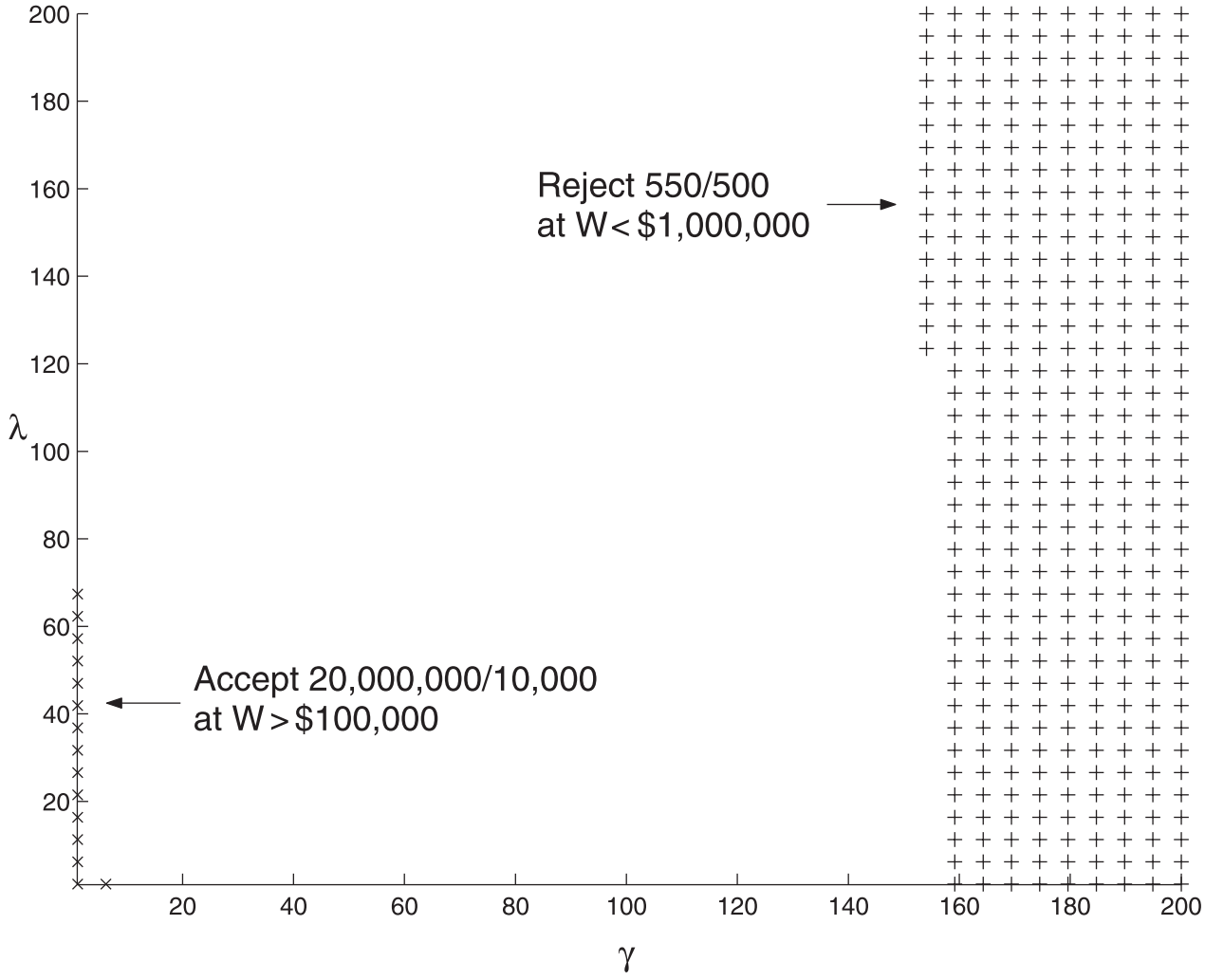


FIGURE 1. ATTITUDES TO MONETARY GAMBLES FOR AN AGENT WITH FIRST-ORDER RISK-AVERSE PREFERENCES

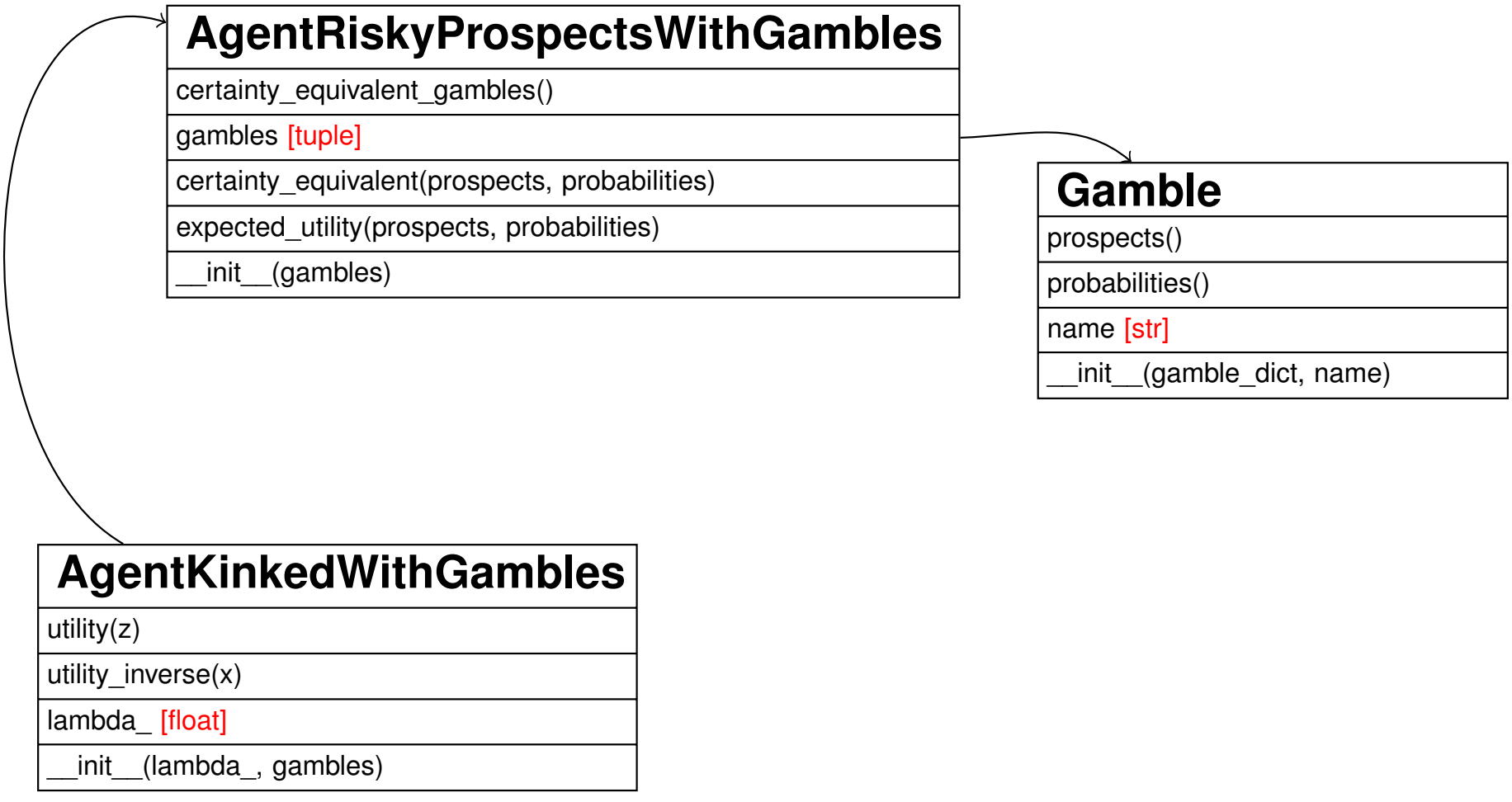
BHT example revisited

- Lots of different agents, two lotteries
- Solution:
  - Vary an agent's preferences
  - Attach a tuple of `Lottery` instances as attributes

Horizontal relationships

- Inheritance: Vertical relationship
- Attaching classes as attributes: Horizontal relationship
- Classic example is from bookkeeping
  - Class `Person`: Age, name, address, ...
  - Class `Address`: Street, number, zip code, town, ...
  - A person could have one or multiple addresses

Horizontal relationships



```
class Gamble:

    def __init__(self, gamble_dict, name):
        self._set_prospects_probabilities(gamble_dict)
        self.name = name

    def prospects(self):
        """Return the prospects as a Mx1 NumPy array."""
        return self._prospects

    def probabilities(self):
        """Return the probabilities as a Mx1 NumPy array."""
        return self._probabilities

    def _set_prospects_probabilities(self, gamble_dict):
        self._prospects = np.array(list(gamble_dict.keys()))
        self._probabilities = np.array(list(gamble_dict.values()))
        self._check_probabilities()

    def _check_probabilities(self):
        assert np.sum(self._probabilities) == 1.0

    def __str__(self):
        return "Gamble instance: {}".format(self.name)
```

```
class AgentRiskyProspectsWithGambles:

    """Superclass for an agent who makes utility evaluations of risky
    prospects, which can be attached to the agent."""

    def __init__(self, gambles):
        self._set_gambles(gambles)

    def certainty_equivalent_gambles(self):
        """Return a dictionary with "name: certainty equivalent"
        entries for the gambles attached to the class instance."""
        out = {}
        for g in self.gambles:
            out[g.name] = self.certainty_equivalent(
                g.prospects(),
                g.probabilities()
            )
        return out

    def _set_gambles(self, gambles):
        if isinstance(gambles, Gamble):
            self.gambles = (gambles,)
        elif isinstance(gambles, list) or isinstance(gambles, tuple):
            self.gambles = tuple(gambles)
        else:
            raise ValueError(
                'Must pass "gambles" argument as "Gamble" instance,' +
                ' list, or tuple.'
            )
        self._check_gamble_names()
```

Horizontal relationships

```
class AgentKinkedWithGambles(AgentRiskyProspectsWithGambles):

    """An agent described by a kinked utility function,
    which only has a loss aversion coefficient *lambda_*.

    """

    def __init__(self, lambda_, gambles):
        self.set_preferences(lambda_)
        super().__init__(gambles)


    def set_preferences(self, lambda_):
        """Set the preference parameter and make a domain check."""
        assert lambda_ > 0, 'Loss aversion must be positive.'
        self.lambda_ = lambda_
```

Horizontal relationships

```
# Define the gambles.
small_stakes = Gamble({-500: 0.5, 550: 0.5}, 'small')
large_stakes = Gamble({-10000: 0.5, 20000000: 0.5}, 'large')
# Set up the agent instance.
gyro_gearloose = AgentKinkedWithGambles(1.0, [small_stakes, large_stakes])
# Evaluate gambles.
print(gyro_gearloose.certainty_equivalent_gambles())
# Change preference parameter
gyro_gearloose.set_preferences(1.5)
# Evaluate gambles.
print(gyro_gearloose.certainty_equivalent_gambles())
```

```
{'small': 25.0, 'large': 9995000.0}
{'small': -66.66666666666657, 'large': 9992500.0}
```

Vertical vs. horizonal relationships

- Which structure to choose depends on the kind of relationship between two objects
  - `AgentKinked` is said to have a **is-a relationship** with `AgentRiskyProspects`
  - `AgentKinkedWithGambles` is said to have a **has-a relationship** with `Gamble`
- Rule of thumb: When it is obvious to think of the relationship of two classes as being vertical in nature, use inheritance, else use an attribute

Looking back at the big picture
Classes

- Looking at classes as data structure, they provide (yet) more scope for abstraction than dictionaries, as they store methods along with variables
- Compared to an approach that relies on functions alone, this can considerably simplify interfaces if some function parameters stay (mostly) constant
  - Define them upfront as class attributes, and re-use them over and over instead of passing them to the function each time
  - Like we did with the utility function parameters

Looking back at the big picture
Object-oriented programming

- Object-oriented programming minimises code duplication and flexibility by allowing for . . .
  - **Inheritance:**

    Define common characteristics upfront and use them in various subclasses
  - **Polymorphism:**

    Simply exchange objects that share a common interface but have different implementations

Looking back at the big picture
Programming paradigms

- Imperative: Do this, do that!
  - Deprecated
- Declarative: Tell the program what you want to be done
  - Encompasses most of the following paradigms
- Function-oriented: Try to maximise code re-use by putting code used in various places into a function
  - What we have been trying to do until this lecture

Looking back at the big picture
Programming paradigms

- Object-oriented: Think of your programs as interacting objects, which carry data along with methods operating on these data. Maximise abstraction and flexibility by encapsulation, polymorphism, and inheritance
  - Changing his preferences meant changing the state of `gyro_gearloose`
  - Often you'll find the definition that OOP minimises state by encapsulating moving parts
- Functional: Minimise state by relying on functions as much as possible
  - Substitute / complement to OOP, not covered here

When to rely on functions, when on objects?

- OOP is a means to an end, not a goal
  - If your class has two methods, one of which is `__init__`, a function will usually be enough
- Find yourself calling different functions with largely similar sets of arguments?
  - Classes **may** be a good idea
  - Store the common data, simplify functions' interfaces
- Need several states at once?
  - E.g. individuals with different preferences
  - Classes **may** be a good idea
  - Single state: Module suffices

References I

Campbell, Jennifer, Paul Gries, Jason Montojo, and Gregory V. Wilson (2009). *An Introduction to Computer Science Using Python*. Ed. by Daniel H. Steinberg. Practical Programming. The Pragmatic Programmers.

Langtangen, Hans Petter (2009). *A Primer on Scientific Programming with Python*. Berlin and Heidelberg, Germany: Springer.

Acknowledgements

- This course is designed after and borrows a lot from the `Software Carpentry` course designed by `Greg Wilson` for scientists and engineers.

- The Software Carpentry course material is made available under a `Creative Commons Attribution License`, as is this course's material.

License for the course material

[Links to the full `legal` text and the source text for `this` page.] You are free:

- **to Share** to copy, distribute and transmit the work
- **to Remix** to adapt the work

Under the following conditions:

- **Attribution** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

With the understanding that:

- **Waiver** Any of the above conditions can be waived if you get permission from the copyright holder.
- **Public Domain** Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- **Other Rights** In no way are any of the following rights affected by the license:
  - Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
  - The author's moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to `this` web page.