Effective Programming Practices for Economists

6. Basic Programming

Hans-Martin von Gaudecker

Department of Economics, Universität Bonn

universitätbonn

Plan for this lecture

- Why are there no simple programming recipes?
- Getting started with Python & Spyder
- First and main abstraction: Variables
- Code style and annotation, docstrings
- More abstractions
  - Modules and namespaces
  - Conditionals
  - Loops
  - Functions

Programming is . . .

. . . what you do when you can't find an off-the-shelf tool to do what you want

- New estimator? Tailored solution algorithm for model?
- Specifying your project's workflow

- Even if it exists, the cost of finding it may be too high
  - Can't google by semantics

Why is programming hard to teach and learn?

Trying to convey three things at once:

1. Here's something interesting that you might want to do
2. Here's the syntax of whatever language we're using
3. Here are some key ideas about programs and programming that will help you do things.

Why is programming hard to teach and learn?

1. is what engages your interest
2. is what you'll grapple with
   - But you will need to master it in order to do 1.
3. is what's most useful in the long term
   - But it's hard or impossible to learn the general without first learning some specifics
   - Later you'll have deadlines: Getting that graph plotted immediately is more important than the big picture

Before we dive in

What **is** a program?

Answers:

Cynical	“Anything that might contain bugs”
Classic	“Instructions for a computer”
We add	“. . . that a human being can understand”

Turning the things we write (math, human language) into instructions a computer executes takes **a lot** of work

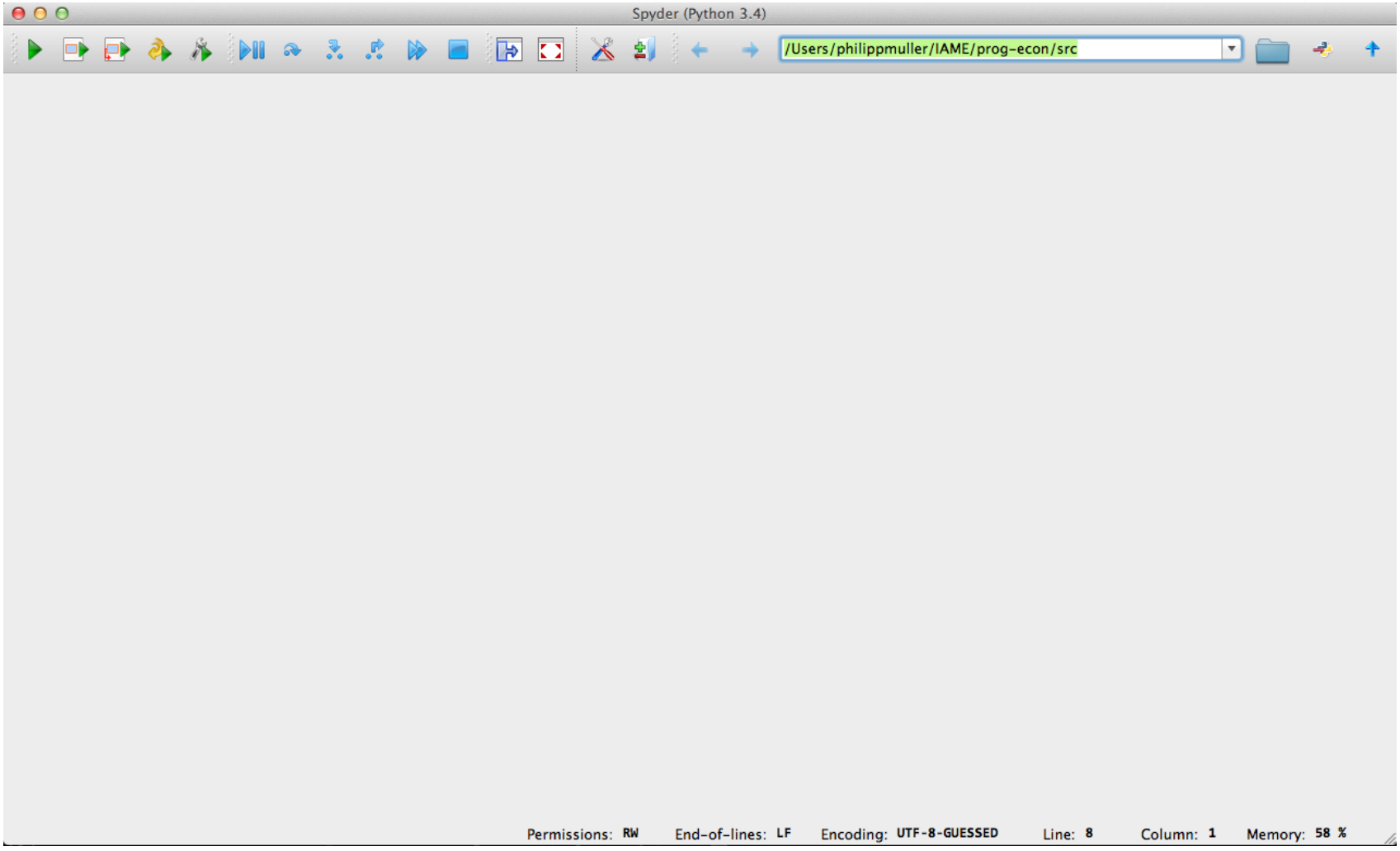
Before we dive in

Relevant measure for “a lot” is time to solution, which equals:

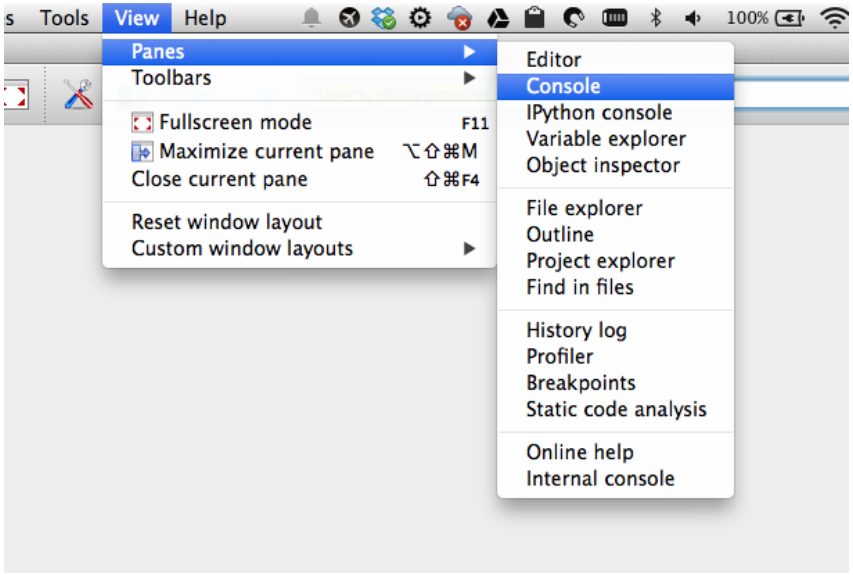
(time to write correct code) + (time for that code to execute)

- The latter halves every 18 months (Moore’s Law)
- The former depends on human factors that change on a scale of years (languages) or generations (psychology)
- Both depend on the programming language chosen
- Scaling is not straightforward ↗

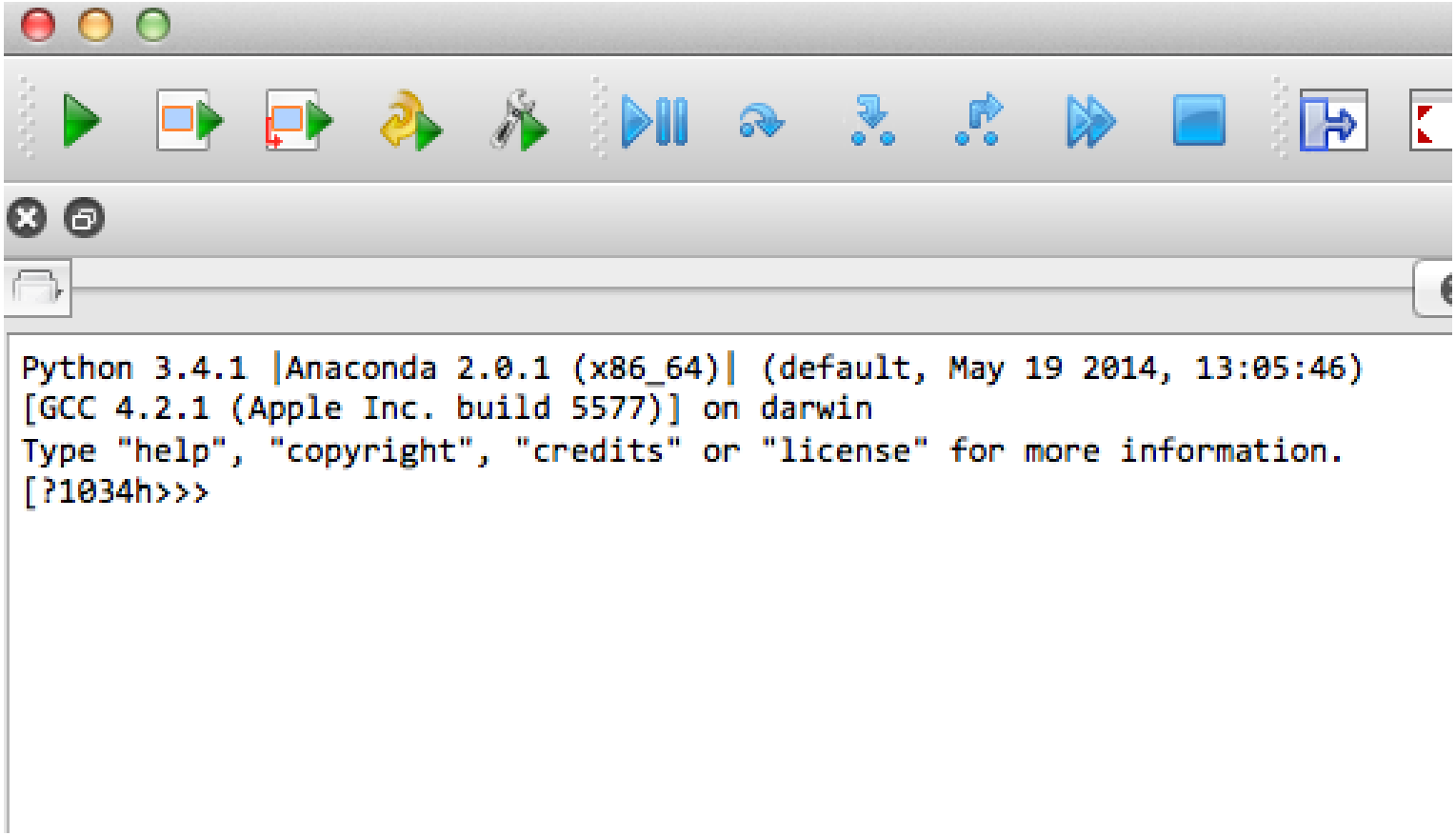
Enabling the console view



Enabling the console view



Opening a Spyder console



Programs store data and do calculations

- Use variables to store data
- Write instructions that use those variables to describe calculations

A first problem

- Calculate the CRRA expected utility evaluation of:

$$\mathcal{L} = \{(0.5, -500), (0.5, 550)\}$$

- Mathematical formula:

$$V(\mathcal{L}, \gamma) = \frac{1}{2} \cdot \frac{(-500)^{(1-\gamma)}}{1-\gamma} + \frac{1}{2} \cdot \frac{550^{(1-\gamma)}}{1-\gamma}$$

A first problem

```
gamma = 2

utility_evaluation = (
    0.5 * (-500) ** (1 - gamma) / (1 - gamma) +
    0.5 * 550 ** (1 - gamma) / (1 - gamma)
)

print('Utility evaluation:', utility_evaluation)
```

Some observations

- Created `gamma` by assignment
- Must create a variable before using it
- A command ends where a line ends
- Two ways to break commands into multiple lines:

```
(code
 in parentheses)

backslash \
character
```

  - Use whatever is easier to read in the particular context
  - Limit lines to a maximum of 99 characters this way

Some observations

- `print()` displays values:
  - Put text (character strings) in quotes
  - `print()` puts a space between values, and ends the line
- Use either single or double quotes to delimit strings
  - Strings must start and end with the same kind of quote
  - Tripling any of the two allows for multiple-line strings:

```
>>> print('''Hello,
... world.'''
... )

Hello,
world
```

Variables don't have types, but values do

- Equivalent C code (a statically typed language):

```c
#include <stdio.h>
#include <math.h>

int main() {
    double gamma, utility_evaluation;

    gamma = 2;

    utility_evaluation = (
        0.5 * pow(-500, 1 - gamma) / (1 - gamma) +
        0.5 * pow(550, 1 - gamma) / (1 - gamma)
    );

    printf("Utility evaluation: %g\n", utility_evaluation);
}
```

- Very different trade-offs in time-to-solution equation 

Variables don't have types, but values do

- Python (dynamically typed) does not require this:

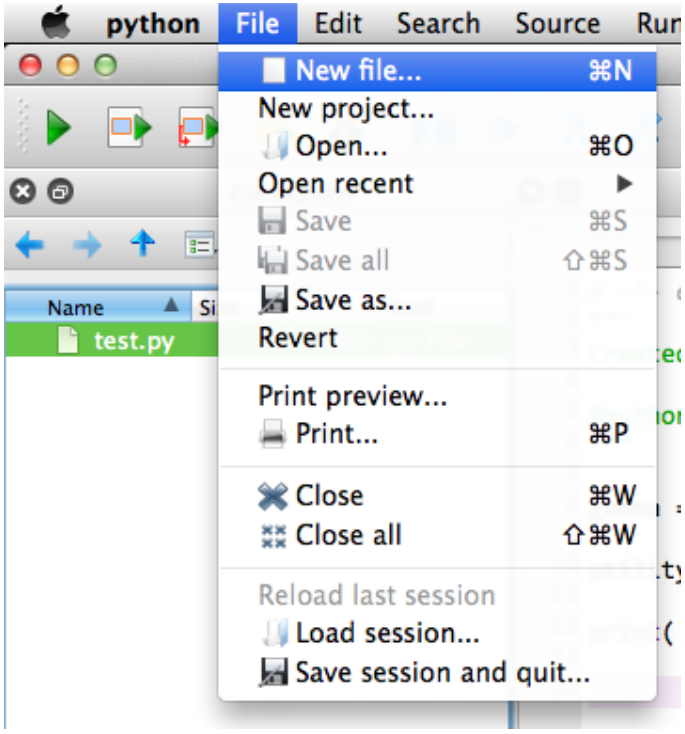
```
gamma = 'two'

gamma = 2
```

- Variables can hold different objects at different times

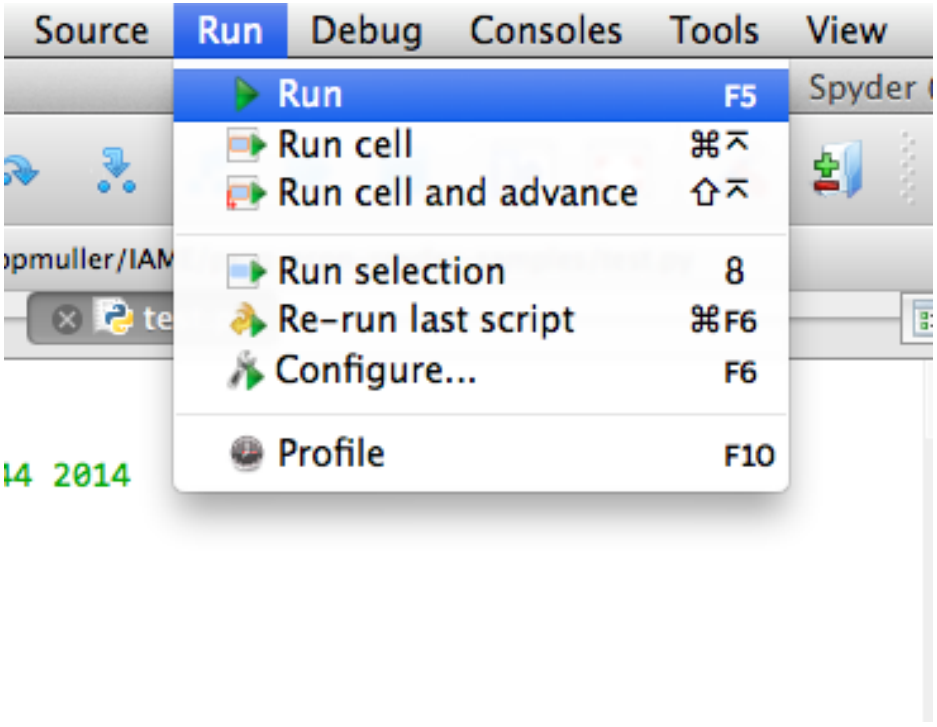
Python source files

- Save code in a file with a .py extension, and . . .
  - type `python file_name.py` in a shell;
  - double-click on `file_name.py` (if associations are set this way);
  - select Run $\rightarrow$ Run in Spyder (but learn keyboard shortcuts = F5)
- A file with code in it is called ‘script’ or ‘module’ in Python
- In the Spyder Package Explorer, right-click on folder where you want to create the file $\rightarrow$ New $\rightarrow$ File . . .

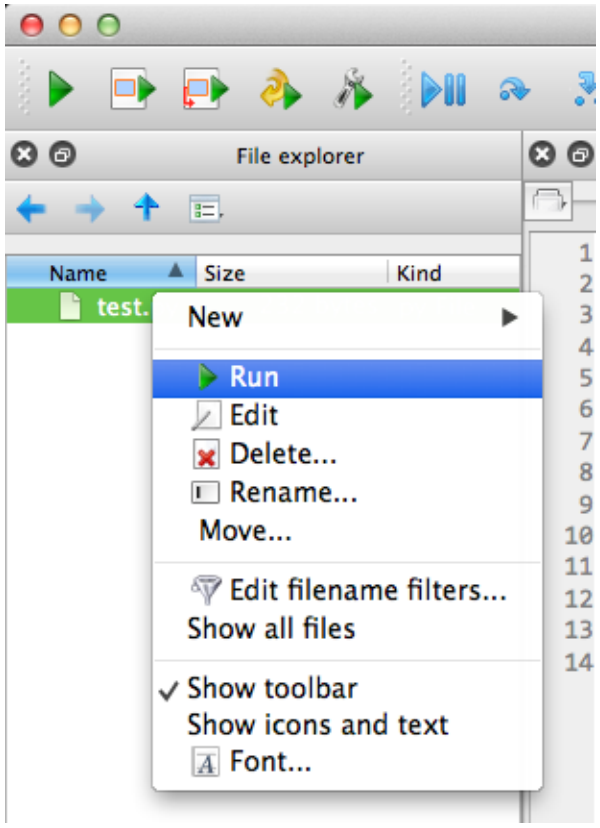
Python source files



Python source files



Python source files



A first generalisation

```
"""Compute the utility value of a lottery under CRRA utility.
"""
# Define the characteristics of the individual.
# Define the characteristics of the lottery.
# Compute the expected utility value.
# Display the results.
```

Comments

- Anything from '#' to the end of the line is a comment
- One of two ways to document our code
  - Explain in human-friendly language what it does
- E.g. explain γ in detail (useful in larger project):

```
# Define the Arrow-Pratt coefficient of relative risk aversion
gamma = 2
```

- Even better: Call it `coefficient_rra`
- Conventions are important, but ...
  - `http://www.python.org/dev/peps/pep-0008/`

A Foolish Consistency is the Hobgoblin of Little Minds

[. . .] code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. [. . .]

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important

But most importantly: know when to be inconsistent – sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Style guide for comments

- Put them before the code you describe
- Use full English sentences, including correct punctuation and capitalisation
  - Unless comment is **really** short (capitalisation still applies)
  - Do not change case of identifiers (e.g. variable names)
- Use inline comments sparingly, do not state the obvious

```
x = x + 1 # Increment x
```

- **Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!**

Docstrings

- A docstring is a string literal that occurs as the first statement in a module, . . .
  - `http://www.python.org/dev/peps/pep-0257:`
- Accept as is for now, treat like a comment
- Describe what the module is doing

Style guide for docstrings

- Use three double quotes, even if it is a single line
- If it spans multiple lines, terminate it by an empty line before the closing quotes
- The docstring prescribes the module's effect as a command ("Do this", "Return that"), not as a description; e.g. don't write "Returns the utility . . . ."
- Use docstrings to describe "the big picture"
- Use comments to describe implementation details which are not immediately clear

Style guide: Naming Conventions
Descriptive: Naming Styles

Example	Description (if relevant)
`b` `B`	Single lowercase letter Single uppercase letter
`lowercase` `lower_case_with_underscores`	
`UPPERCASE` `UPPER_CASE_WITH_UNDERSCORES`	
`CapitalizedWords`	Or CapWords, or CamelCase (so named because of the bumpy look of its letters). This is also sometimes known as StudlyCaps. Note: When using abbreviations in CapWords, capitalize all the letters of the abbreviation. Thus CRRA-Parameter is better than CrraParameter
`mixedCase`	Differs from CapitalizedWords by initial lowercase character
`Cap_Words_With_Underscores`	Ugly!

Style guide: Naming Conventions
Prescriptive: Variables and constants

- [Variable] names should be lowercase, with words separated by underscores as necessary to improve readability

- Constants are usually declared on a module level and written in all capital letters with underscores separating words. Examples include `MAX_ITERATIONS` and `TOTAL`
  - Use constants to avoid “magic numbers” (hard-coded)

DRY **Don’t repeat yourself**

Style guide: Naming Conventions
Prescriptive: Variable names

- Variable names should be long enough to be self-explicatory
  - `gamma` and `v` are okay here, but not if you want to use them in a larger context
- Rule of thumb: Keep the length of variable names inversely proportional to the scope where they are used
  - Need a counter for elements in a container, relevant in 5 adjacent lines: `i` is great
  - Coefficient of relative risk aversion used at various places in a 2000-line project: `coefficient_rra` tells you exactly what the variable contains

My first real module

```
"""Compute the utility value of a lottery under CRRA utility.

"""

# Define the parameters of the utility function.
gamma = 2

# Define the characteristics of the lottery.
prob_low = 1 / 2
z_low = -500
z_high = 550

# Compute the expected utility value.
utility_evaluation = (
    prob_low * (z_low ** (1 - gamma)) / (1 - gamma) +
    (1 - prob_low) * (z_high ** (1 - gamma)) / (1 - gamma)
)

# Display the results.
print('Utility evaluation:', utility_evaluation)
```

Making the code more general

- What happened again at $\gamma = 1$ to:

$$u(z, \gamma) = \frac{z^{1-\gamma}}{1-\gamma}$$

-

- Implement with conditional statement

Conditionals

- General syntax:

```
if x == 0:
    print('x is zero')
elif x > 0:
    print('x is greater than zero')
else:
    print('x is less than zero')
```

- Use double equals sign for equality testing
  - Else you will get a `SyntaxError` – the single equals sign is only the assignment operator

Conditionals

- The expression between `if / elif` and the colon is evaluated according to standard Boolean logic
  - Short-circuit
- Could also be an object, which is automatically truth-tested
  - `if 0:` evaluates to `False`
- Read `http://docs.python.org/3/library/stdtypes.html` up to section 4.4.0 inclusive for operators and numeric types
  - Keep it under your pillow

Whitespace matters

- Why not `begin / end` or `{...}` ?
  - Studies have shown that indentation is what people pay attention to
  - Go make sense of `prog1_20031011.m` supplied with BHT at `http://dx.doi.org/10.1257/aer.96.4.1069`, which has wild indentation ...
  - Or almost any non-trivial Stata code

Whitespace matters

- Use four spaces
  - **The** most widespread convention, no reason to break it
  - Although it only has to be consistent within each block
  - **No tabs!** (convention, again – but editor display problems)
- Does not matter in continued lines
- Spyder handles most cases automatically

```
"""Compute the utility value of a lottery under CRRA utility.

"""

from math import log


# Define the characteristics of the individual.
gamma = 1.0
endowment = 10000.0

# Define the characteristics of the lottery (consolidated with endowment).
prob_low = 0.5
z_low = -500.0 + endowment
z_high = 550.0 + endowment

# Compute the expected utility value.
if gamma == 1.0:
    utility_evaluation = (
        prob_low * log(z_low) +
        (1 - prob_low) * log(z_high)
    )
else:
    utility_evaluation = (
        prob_low * (z_low ** (1 - gamma)) / (1 - gamma) +
        (1 - prob_low) * (z_high ** (1 - gamma)) / (1 - gamma)
    )

# Display the results.
print('Utility evaluation:', utility_evaluation)

Utility evaluation: 9.211464108246421
```

Comments on previous code

- Need to import many math-specific things explicitly
  - Matlab assumes you mean the logarithm by $\log(\cdot)$
  - For a web programmer, the intuitive meaning might be keeping access logs to a website
  - $\Rightarrow$ Different namespaces
- Good practice to define a floating point number explicitly when you require one: `x = 1.0` instead of `x = 1`

Comments on previous code

- Careful with value testing and floating points
  - Rounding errors
  - Unproblematic here
  - Different story when you must hit the exact value
- How to calculate $V(\mathcal{L}, \gamma)$ for different values of γ?

Repeating operations

- Computers are useful because they can do lots of calculations on lots of data
- We need a concise way to represent multiple values and multiple steps
  - Writing out 1M additions would take longer than doing them
- Use loop to perform multiple operations
  - Like $\sum$ in mathematics
  - But we have to break it down into sequential steps

General syntax of `for`-loops

```
for running_variable in iterable:

    do_something

    # Stop the current iteration step
    # and go to the top of the loop.
    if condition_1:
        continue

    # Stop the current iteration step
    # and go to the first statement after the loop.
    if condition_2:
        break

    # The next iteration step starts
    # where the indentation ends.

first_statement_after_the_loop
```

Functions

- Our code is pretty hard to read now
- In the body of the `for`-loop, need to:
  - keep track of what the `for`-statement is doing
  - remember the values for $p_{\text{low}}, z_{\text{low}}, z_{\text{high}}, \gamma$
  - understand how $u(z, \gamma)$ and $V(\mathcal{L}, \gamma)$ are calculated
  - realise that $u(z, \gamma)$ appears 2×2 times
- What if we wanted to calculate $u(z, \gamma)$ or $V(\mathcal{L}, \gamma)$ at various places of a larger program?

Functions

DRY **Don't repeat yourself!**

- Whenever you find yourself copying and pasting code, make a function out of it (at the minimum put it in a loop)
- Else, if you change something in your code you will most certainly forget to change it in at least one other place
- But remember: A foolish consistency ...

- Design choice:
  - Write a function for $u(z, \gamma)$?
  - Or for $V(\mathcal{L}, \gamma)$?

- Eventually both, but start with $u(z, \gamma)$

```
def u(z, gamma):
    """Return the utility value of *z* under CRRA utility with
    Arrow-Pratt coefficient of relative risk aversion *gamma*,
    allowing for log utility.

    """

    if gamma == 1.0:
        return log(z)
    else:
        return z ** (1 - gamma) / (1 - gamma)


... [set the parameters as before] ...


for gamma in gamma_1, gamma_2:
    # Compute the expected utility value.
    utility_evaluation = (
        prob_low * u(z_low, gamma) +
        (1 - prob_low) * u(z_high, gamma)
    )

    # Display the results.
    print('Utility evaluation:', utility_evaluation)
```

Comments on previous code

- Syntax for function definition:

```
def function_name(argument_1, argument_2, ...):
```

  - Use empty parentheses for functions without arguments
  - Same style conventions for `function_name` as for variables

- Docstring prescribing what the function should do
  - Same rules as for modules
  - Same indentation as for conditionals
  - Convention: Start on the line after the function definition

Comments on previous code

- Function body:
  - Indented block until function ends
  - `return expression` is optional, can occur multiple times
  - If `return` is specified without an argument, or left out, the function returns `None`
- Need to define function before it is used
  - Just as with variables

`None` indicates the absence of any value, evaluates to `False` in truth value testing

More on functions

- When you invoke a function, provide arguments by order:

```
v = u(z_high, gamma)
```

- Or provide them by name:

```
v = u(z=z_high, gamma=gamma)
```

- Can start with order, continue with names:

```
v = u(z_high, gamma=gamma)
```

- But not vice-versa

- When provided by name, the order can be arbitrary

More on functions

- Set default values in definition with:

```
argument=default_value
```

- Will be used when no value is supplied for that argument

- E.g. assume risk neutrality if γ is not specified:

```
def u(z, gamma=0.0):
    """Return the utility evaluation ..."""
```

- Functions nest (keep in mind, don't use much)

Example with functions for u and v

```
def u(z, gamma):
    """Return the utility value of *z* under CRRA utility with
    Arrow-Pratt coefficient of relative risk aversion *gamma*,
    allowing for log utility.

    """

    if gamma == 1.0:
        return log(z)
    else:
        return z ** (1 - gamma) / (1 - gamma)


def v(p_low, z_low, z_high, gamma=0.0):
    """Return the utility evaluation of a two-outcome gamble described
    by *p_low*, *z_low*, *z_high* under CRRA utility with Arrow-Pratt
    coefficient of relative risk aversion *gamma*.

    The default value for *gamma* implies risk neutrality.

    """

    return p_low * u(z_low, gamma) + (1 - p_low) * u(z_high, gamma)
```

A closer look at tracebacks

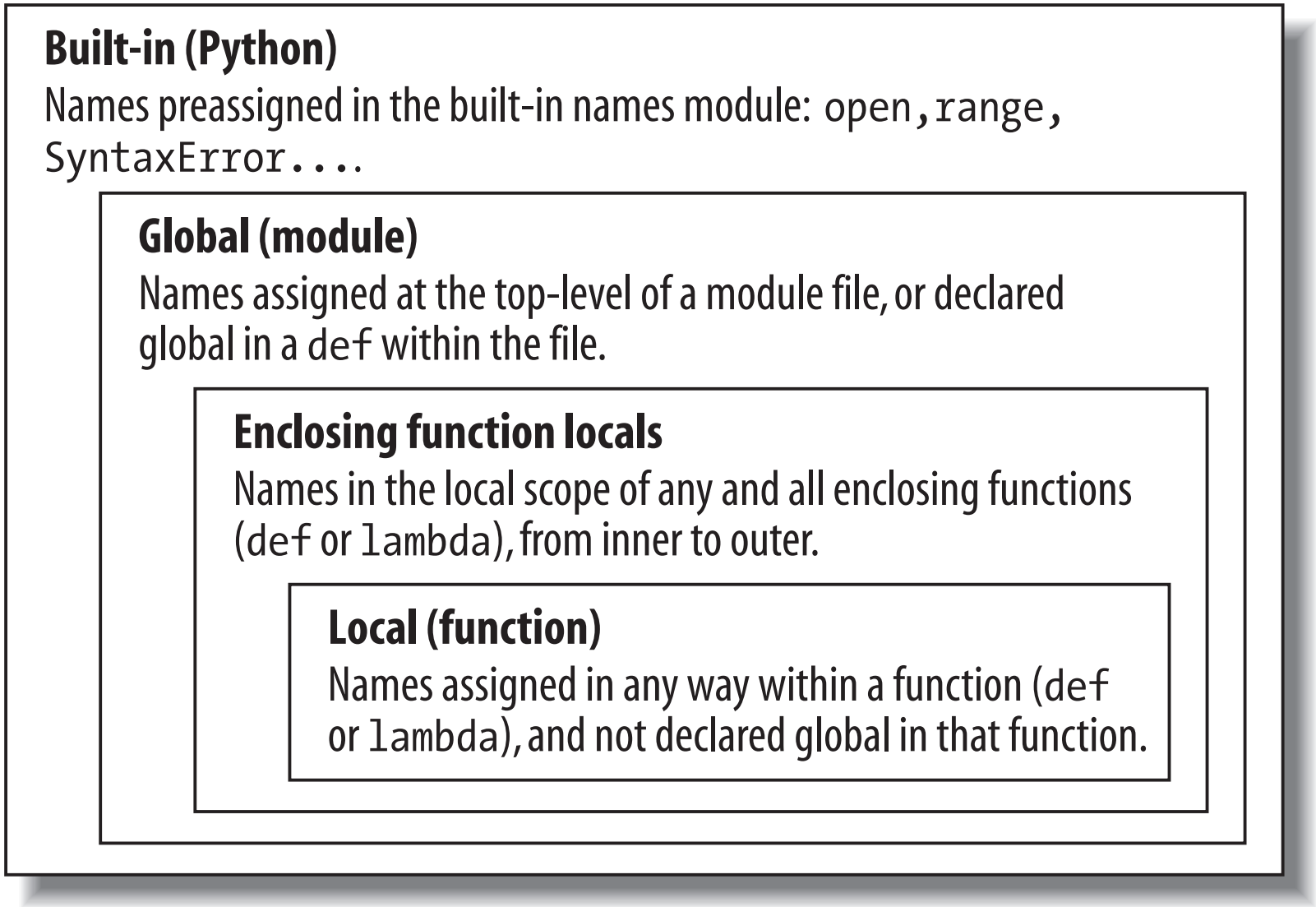
Result after inserting `produce_name_error` as the first line of `u(.)` in previous code:

```
Traceback (most recent call last):
  File "utility_crra_functions.py", line 46, in <module>
    utility_evaluation = v(prob_low, z_low, z_high, gamma)
  File "utility_crra_functions.py", line 30, in v
    return p_low * u(z_low, gamma) + (1 - p_low) * u(z_high, gamma)
  File "utility_crra_functions.py", line 14, in u
    produce_name_error
NameError: name 'produce_name_error' is not defined
```

Namespaces

- Objects (variables, functions) assigned within a function are called **local** to this function
  - You cannot reference (see) them outside the function
- A function has access to objects defined at “higher” levels
  - Those defined in the **enclosing scope** and built-ins
  - Those objects are called **globals**
  - You can also declare global variables in a function, useful in a very limited number of cases
- Resolve names from inside out – the LEGB rule

The LEGB rule (Lutz, 2007, p. 313)



Sharing code across modules

- The statement

```
from other_module import object
```

  puts `object` from `other_module.py` into the current namespace

- We saw that before:

```
from math import log
```

  put the function `log` from the built-in module `math` into the global namespace of our `utility_crra` module

Which modules does Python know about?

- The search path always includes the directory where the current module lives

- The `path` attribute of the built-in module `sys` has the full search path:

```
>>> from sys import path
>>> for item in path:
>>>     print(item)
```

```
/Users/[...]/anaconda/lib/python34.zip
/Users/[...]/anaconda/lib/python3.4
/Users/[...]/anaconda/lib/python3.4/plat-darwin
/Users/[...]/anaconda/lib/python3.4/lib-dynload
[...]
```

Which modules does Python know about?

- System-wide additions are put in `site-packages` directory
- Packages are a structured way of storing modules in a hierarchical structure. Usage:

```
from scipy.optimize import newton
```

- See the internals later in the course

Different import syntaxes

- What if we want to have access to all 564 objects provided by NumPy?
- Shortcut is:

```
from numpy import *
```

- But how would we ever find our own variables?
- And what if we wanted to define an object `unique` which exists in NumPy? (or did so accidentally)

Different import syntaxes

- Solution:

```
import other_module
```

- Then access objects defined in `other_module` by:

```
other_module.object
```

- Terminology: `object` is an **attribute** of `other_module`

- After `import matplotlib`, you would have to type those 10 characters all the time

- You can only access it by typing `mpl` after `import as`:

```
import matplotlib as mpl
```

Review of concepts

- Variables, strong vs. dynamic typing
- Style guide / coding conventions
- Comments and docstrings
- Control flow and indentation
  - If-statements
  - Loops
- Functions
- Namespaces, modules, and imports

Further material

- Official Python tutorial / documentation

  `http://docs.python.org/3/`

- Accessible introduction for doing useful non-science stuff with Python

  Sweigart (2015)

- Interactive Python book
  - `http://interactivepython.org/runestone/static/thinkcspy/index.html`

- Software Carpentry Python intro
  - `http://swcarpentry.github.io/python-novice-inflammation/`

References I


Lutz, Mark (2007). *Learning Python*. 3rd ed. O'Reilly Media.


Sweigart, Al (2015). *Automate the Boring Stuff with Python*. San Francisco, CA: No Starch Press.

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.