

# Python 3 for scientific computing

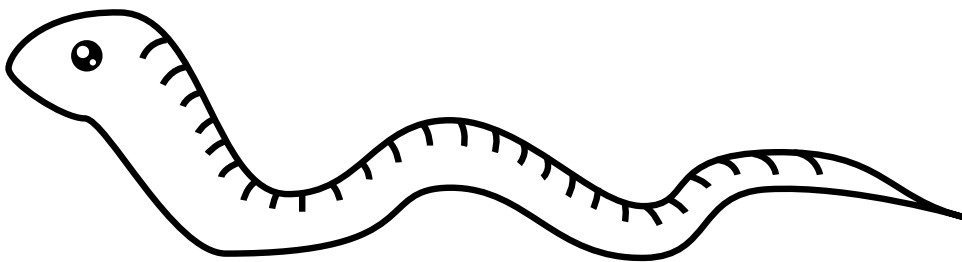
Lecture 3, 7.2.2018

A second practical look at Python 3

Juha Jeronen

[juha.jeronen@tut.fi](mailto:juha.jeronen@tut.fi)

U+29, RIGHT PARENTHESIS



Spring 2018, TUT, Tampere  
RAK-19006 Various Topics of Civil Engineering



TAMPERE  
UNIVERSITY OF  
TECHNOLOGY

# Meta

## About the code examples

- The code examples in these slides are meant to be copy'n'pasteable so that they can be easily tried out in Spyder (in either a script or the REPL).
- Double-quotes might not always work, due to the slides being prepared in an office suite, which very eagerly replaces them with “smart quotes”:
  - “ = U+8220 = LEFT DOUBLE QUOTATION MARK
  - ” = U+8221 = RIGHT DOUBLE QUOTATION MARK
  - " = U+34 = QUOTATION MARK
- U+34 is the one used in programming; it appears when you type a double-quote on the keyboard in almost any software, *except office suites*.
- Copy'n'pasting code examples from the lecture *notes* does not work due to an issue with LaTeX and the *verbatim* environment.
  - <https://tex.stackexchange.com/questions/62221/ensure-verbatim-code-block-is-copy-paste-able>
- Some code examples as .py files are available on GitHub:  
<https://github.com/Technologicat/python-3-scicomp-intro/tree/master/examples>

# Getting to know Python (1 of 2)

*Previously on Python 3 for scientific computing...*

- Python's documentation
- Basic control flow (loops, conditionals)
- Containers
  - Basic operations on lists and strings
  - String formatting
  - Slicing (indexing operation)
  - tuple (immutable list)
  - zip (interleaving iterables)
  - Tuple unpacking (destructuring bind and its friends)
  - set, dict
  - List comprehension
- Functions
  - Defining; returning values
  - Argument passing (by position, by name)
  - Argument default values
  - Docstrings (where all the *help* comes from)
  - Functions as first-class objects; **lambda** (anonymous func.)
- Getting started with Spyder IDE (for exercises)

[ ] ( )  
{ }

%0.3g

"a" 'a'  
"""a"""

B = [f(x) for x in A if g(x)]

# Getting to know Python (2 of 2)

## (This set of slides)

- Modules: defining, loading (**import**)
- Advanced control flow: *exceptions* (**try**, **except**, **finally**)
- The **with** statement (ensuring open files are eventually closed)
- Python and Unicode
- Language features, a closer look by example
  - Names and values (Python's "variables")
    - Special value **None**, keyword **is**; **global** and **nonlocal**
  - Paradigms: *imperative* vs. *functional* programming
  - Lexical scoping (contrast dynamic scoping)
  - Type system: strong typing, duck typing
  - Call-by-sharing (argument passing model)
- How to structure Python programs
  - Script (just a sequence of statements or expressions)
  - Procedural program
    - Conditional execution of the main program  
(allows dual use as main program and library)
  - Object-oriented program
    - Basics of object-oriented programming (OOP)  
(or, how to get rid of global variables in your solvers!)
    - Inheritance (single and multiple)

Ordering  
of topics  
swapped

# Modules

## Defining

- In Python, code is organized into *modules*.
- *Each .py file is a module*. Hence, **to create a module, just save your work**.
  - Basically no restrictions on module naming (contrast Java, class Foo  $\Rightarrow$  Foo.java).
  - The module filename is independent from the symbol names it exports (contrast MATLAB).
- A module *exports* all names defined in its top-level scope, that do not begin with a single underscore (`_`). Beginning a name with an underscore marks it as private to the module.
- There are some magic names that begin and end with two underscores.
  - A notable one is `__name__`, which Python automatically creates at runtime, and fills with the module name (based on the filename), as a string. The main program has name “`__main__`”.
  - A standard convention is to assign your module version into `__version__`, as a string. E.g. NumPy and many other libraries do this, so programs can easily check which version of a particular library is installed.
  - You can define `__all__` in your module to restrict which names star-imports (see below) will actually import from your module. If not defined, the default (usually fine) is all exports.

```
__all__ = ['answer', 'f']  # names that will be imported by star-importing this module
answer = 42
__private = 'Ryan'  # begins with a single underscore, not exported
def f():
    pass  # pass is Python for “do nothing”; often used in small examples like this
def g():
    pass  # “g” not in __all__  $\Rightarrow$  not imported by star-imports, but visible to all other imports
```

# Modules

## Loading (**import**)

- To tell your program to load (run) another file called *something.py*, use the **import** statement:

```
import something    # module name, without the .py
```

- Importing allows you to reuse your functions and/or classes.
  - But of course, you don't then want the imported module to run its main program. When we talk about structuring programs in Python, we will see how to avoid that.
- In any particular session, the top-level code in each module is executed only once, even if the same module is **imported** multiple times.
  - This is obviously a performance optimization, but also has a use for creating your own singleton object instances. Since the top-level runs only once...
  - If a specific need arises, one can force Python to reload a given module. In Python 3.4+:

```
import importlib  
importlib.reload(something)
```

<https://stackoverflow.com/a/4113189>

- For more on modules, see the official tutorial:

<https://docs.python.org/3/tutorial/modules.html>

# Modules

## Variants of `import`

- **`import`** something
  - Access *foo* defined in *something* as *something.foo*
  - Where each name comes from is clearly visible at the use site.
  - Very pythonic. Probably the most common.
- **`from`** something **`import`** foo, bar, baz
  - This is called a “from-import”.
  - *something.foo* becomes just *foo* at the import site; similarly for *bar*, *baz*
  - Requires anyone reading the code to spot the from-import.
  - Also acceptable.
- **`from`** something **`import`** \*
  - This is called a “star-import”.
  - It inserts **all names** from *something* into the scope at the import site.
    - By default, “all” means “anything not beginning with a single underscore”. The custom list `__all__`, if it is defined in the module being imported, overrides this.
  - Please only do this in an interactive session – Python is not MATLAB.
    - In scripts and programs, star-imports significantly harm readability, because it is no longer explicit from which module each used name comes from.
    - Name conflicts may occur, if two or more modules export the same names. For any conflicting names, the most recent star-import wins.
      - E.g. NumPy and SymPy both export *sin*, *cos*, but numerical vs. symbolic!

# Modules

## Variants of **import ... as** – renaming on load

- **import** something **as** otherthing
  - Access e.g. *something.foo* as *otherthing.foo*
  - Acceptable, common. Practically all numerical users do this:

```
import numpy as np  
import matplotlib.pyplot as plt
```

- **from** something **import** foo **as** goo, bar **as** tavern, baz
  - *something.foo* becomes just *goo* at the import site; *something.bar* becomes *tavern*; *something.baz* becomes *baz* (since no **as** part)
  - Useful to avoid name conflicts, if you prefer to use from-imports.
- The above quick summary covers 99% of actual use cases.

For the final 1%, see the documentation:

[https://docs.python.org/3/reference/simple\\_stmts.html#import](https://docs.python.org/3/reference/simple_stmts.html#import)  
<https://docs.python.org/3/reference/import.html>



# Modules

## Packages

- Modules can be organized into *packages*, which mirror the folder structure on the hard drive:

<b>flarbuzz/</b>	← <b>package</b> ; top-level folder of project
__init__.py	← optional in Python 3; runs when <b>import</b> flarbuzz
dostuff.py	← to load this (explicitly), <b>import</b> flarbuzz.dostuff
<b>morebuzz/</b>	← <b>subpackage</b> ; to load this, <b>import</b> flarbuzz.morebuzz
__init__.py	← optional; a subpackage may also have its own init script
foo.py	← <b>import</b> flarbuzz.morebuzz.foo
bar.py	← etc.

- Often the init script star-imports names from the modules of the package (at that level), so that the user will see those names directly in the package namespace (e.g. for a function *quux* defined in *dostuff.py*, the user will see *flarbuzz.quux* instead of *flarbuzz.dostuff.quux*.)
  - This is an acceptable use of star-import, for information hiding of implementation details.
- E.g. “**import** numpy.linalg” loads the *linalg* subpackage of NumPy
- Historically (Python 2), the presence of *\_\_init\_\_.py* (even if a blank 0-byte file) was required to mark a folder as a Python package. In Python 3, no longer required, but still often used.
- More on packages:  
<https://docs.python.org/3/tutorial/modules.html#packages>
- If you make your own packages, read up on *absolute vs. relative import*  
<https://www.python.org/dev/peps/pep-0328/>

# Advanced control flow

## Exceptions: **try**, **except**, **finally**

- **Problem:** in almost any computation, errors *will* eventually occur. How to:
  - pass around the necessary metadata? (E.g. success vs. error; error type and details)
  - structure the source code?
- Classical solution: **Look Before You Leap (LBYL)**

```
if f(foo): # consider an "f" that returns truthy on success, falsey on failure
    print('yay, successful!')
else:
    print('something went horribly wrong')
```

- The code becomes peppered with non-essential **if** statements, and error handling is interleaved with the usual case of successful operation ⇒ unnecessarily hard to read.
- Modern solution, *exceptions*: **Easier to Ask for Forgiveness than Permission (EAFP)**

```
try:
    f(foo)
    print('yay, successful!')
except MyInterestingError as e:
    print('one thing went horribly wrong')
except SomeOtherError as e:
    print('the other thing went horribly wrong')
```

- Roughly, the usual case can be fully written out first, with all the error handlers placed below it.

# Advanced control flow

- Full syntax: Exceptions: **try**, **except**, **finally**

```
try:
    ...
except ErrorType as e: # catch any exception that isinstance(ErrorType)
    ...
except (ErrorType1, ErrorType2, ...) as e: # a handler can catch several alternatives
    ...
else: # Python-specific feature: an else block for a try means “if no exception occurred”
    ...
finally: # always runs, exception or not; runs last, just before the whole try... construct exits
    ...
```

*Exception handler* (points to the `except` block)

That is valid Python, if ErrorType is defined (points to `isinstance(ErrorType)`)

- In sufficiently complex real-world programs, you may need to nest **try** blocks, or split the code in a function into several consecutive **try** blocks, depending on the details of your program.
- The nature of exceptions is that they can happen **anywhere**, and you just have to live with that.*  
—Guido van Rossum, <https://www.python.org/dev/peps/pep-0343/>
- See <https://docs.python.org/3/tutorial/errors.html>  
<https://docs.python.org/3/library/exceptions.html#builtin-exceptions>  
Lecture material, p. 39.
- Exceptions are nowadays a common feature, but terminology and details vary by language:
  - Python: **try**, **except**, **finally**; **raise**; vs. Java: **try**, **catch**, **finally**; **throw**
  - C++ has **try**, **catch**; **throw**; but no **finally**, because it prefers the [RAII design pattern](#).

# Context manager

## The **with** statement

- **Problem:**

```
f = open("my_input.txt", mode="rt", encoding="utf-8")
for line in f:
    do_something(line)  # but wait – what happens if this raises an exception?
f.close()
```

- If an exception occurs, the file is never closed (until Python exits)! ⇒ **resource leak**
  - Strictly speaking, the garbage collector (automatic memory management) will pick up and discard the object at some point after “f” (the only reference to it) has gone out of scope, but no guarantee on when.

- **Solution**, first attempt:

```
try:
    f = open("my_input.txt", mode="rt", encoding="utf-8")
    for line in f:
        do_something(line)
except MyInterestingError as e:
    ... # whatever
finally:
    f.close()  # that should do it, right?
```

But now, what if **open** fails (e.g. **FileNotFoundError**)? The **finally** runs, but no “f” ⇒ **NameError**!

# Context manager

## The **with** statement

- **Solution**, hopefully correct?

```
try:
```

```
    f = open("my_input.txt", mode="rt", encoding="utf-8")
```

```
    try:
```

```
        for line in f:
```

```
            do_something(line)
```

```
    except MyInterestingError as e:
```

```
        ... # whatever
```

```
    finally:
```

```
        f.close()
```

```
except FileNotFoundError as e:
```

```
    print("'my_input.txt': no such file")
```

- Added parts marked with `...` . Additionally, the call to **open** was moved outside the inner **try** block.
- It would sure be nice to abbreviate that!

# Context manager

## The **with** statement

- Enter the **with** statement, introduced in Python 2.5:

```
try:
    with open("my_input.txt", mode="rt", encoding="utf-8") as f: # the as ... part is optional
        for line in f:
            do_something(line)
except FileNotFoundError as e:
    print("'my_input.txt': no such file")
except MyInterestingError as e: # now f is already closed here
    ... # whatever
```

- Upon entering the **with** block, the resource is acquired, and the reference is bound to “f”.
- Upon exiting the **with** block – regardless of whether normally, or due to an exception or *non-local goto* (such as **continue**, **break** inside a loop, or **return**) – the resource is released.
  - In the case of **open**, this means closing the file.
  - Has many other uses; e.g. timing, like MATLAB's tic/toc. See lecture notes, p. 40; code: <https://github.com/Technologicat/python-3-scicomp-intro/blob/master/examples/simpletimer.py>
- This syntax eliminates the *accidental complexity*, leaving only *essential complexity*.
  - The file must be opened, processed, and closed; any interesting exceptions must be caught.
  - [https://en.wikipedia.org/wiki/No\\_Silver\\_Bullet](https://en.wikipedia.org/wiki/No_Silver_Bullet)
- See also:
  - <https://www.python.org/dev/peps/pep-0343/> The “with” statement (2.5+)
  - <http://effbot.org/zone/python-with-statement.htm> Understanding Python's “with” statement

# Python and Unicode

- Roughly speaking, **Unicode** *≈ a single huge codepage for the 21<sup>st</sup> century*
  - So that you can èéüüääö, あ and ∀δ ∃ε:  $|y - x| < \delta \Rightarrow |f(y) - f(x)| < \varepsilon$  in the same sentence!
  - Also ♪ ♫ ♬ ♯ ♪, ♥ ♦ ♣, ☂, ☂, ☂, ☂, ☂, ☂, ♻, ⌨, ☎, ☯, ☹, ✈, ...
  - Standardization of emoji: <http://unicode.org/emoji/charts/full-emoji-list.html>
  - Unicode 10.0: 136,755 characters covering 139 modern and historic scripts, as well as multiple symbol sets. <https://en.wikipedia.org/wiki/Unicode>
- To convert between a character and its *Unicode codepoint* (a natural number):

```
print(ord('λ'))           # 955, decimal, but programmers talk hex
print('U+{:x}'.format(ord('λ'))) # U+3bb  ("U+" common prefix to denote a Unicode codepoint)
chr(0x3bb) # ⇒ 'λ'    (0x... is Python for hex number literal, like in C)
chr(955)   # ⇒ 'λ'    (same number, just expressed in base-10)
'u03bb'    # ⇒ 'λ'    (Unicode escape sequence; \x 2 hex digits, \u 4, \U 8)
'\N{GREEK SMALL LETTER LAMDA}' # ⇒ 'λ'  (escape sequence using character name)
```

- To decrypt a mystery character in Python (if you can copy'n'paste it):

```
from unicodedata import name, lookup
name('λ')           # ⇒ 'GREEK SMALL LETTER LAMDA' (yes, "lamda")
lookup('GREEK SMALL LETTER LAMDA')  # ⇒ 'λ' (lookup is the inverse of name)
```

- Similar chars, e.g. Ω = U+2126, 'OHM SIGN'; Ω = U+3a9, 'GREEK CAPITAL LETTER OMEGA'
- See also:  
[https://en.wikipedia.org/wiki/Universal\\_Character\\_Set\\_characters](https://en.wikipedia.org/wiki/Universal_Character_Set_characters)  
<ftp://ftp.unicode.org/Public/UNIDATA/UnicodeData.txt> (get the names here)

# Python and Unicode

- Motivating factors behind the development of Unicode:
  - Internationalization; need to handle many human languages in the same software.
  - Standardization of many incompatible character sets into a single universal set.
- *Character encoding vs. Unicode*:
  - *Unicode*: roughly, a mapping between (a finite subset of) the natural numbers and the characters in the universal character set; character  $\leftrightarrow$  codepoint.
  - *Encoding* (e.g. ASCII, ISO-8859-1, utf-8, utf-16): a format to store Unicode codepoints (natural numbers!) as a stream of bytes, such as a file on disk.
- Ned Batchelder: *Pragmatic Unicode, or, How do I stop the pain?*  
<https://nedbatchelder.com/text/unipain.html>
  - Make a *Unicode sandwich*:
    - **Decode input** from bytes (e.g. a text file stored as utf-8)
    - Process as **Unicode string** (the string type in Python 3)
    - **Encode output** to bytes (specifying an encoding, such as utf-8)
  - Hence, be (slightly) careful when performing I/O; the rest needs no extra effort.
- See also:
  - <https://docs.python.org/3/howto/unicode.html>
  - Lecture material, pp. 29–30.



# Language features

## Names and values: Python's “variables”

- **Python has no variables**, in the sense of names for fixed storage locations.
  - [https://en.wikipedia.org/wiki/Variable\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Variable_(computer_science))
- Instead, Python has **names** and **values**. (Contrast Fortran, C, C++, ...)
  - Names can be thought of as references or labels. Unlike variables, names have no *type*.
  - Values are object instances. Each value has a type.
  - *Ned Batchelder: Facts and myths about Python names and values*  
<https://nedbatchelder.com/text/names.html>
- For **testing object identity**, Python provides:
  - The built-in function **id**, which returns the memory address of an object instance.
  - The operator **is**, which tells whether two names point to the same object.

```
a = 1
print(id(a))  # 10055552, some seemingly random number
b = a
b is a       # True
```

- What happens here? Recall from lecture 1 (slide 25) that in Python, assignment binds names.
  - To be exact: the assignment “b = a” binds the name “b” to the object instance that is currently bound to the name “a” in the scope where the assignment statement is executed.
- Hence, after the assignment, the names “b” and “a” point to the same object instance.
  - For mutable objects such as lists (and later on this course, NumPy arrays), this is critically important, because often we want to copy a mutable to modify it without affecting the original. For now, keep in mind that **in Python, assignment never copies**.
  - How to copy: `.copy` method of most mutable objects; call the constructor, passing in the original instance; standard library module `copy`. See next slide, and lecture material, p. 49.

# Language features

## Names and values – copying, identity

- Examples of copying a mutable object:

```
lst1 = [1, 4, 9, 16, 25]
lst2 = lst1.copy()           # explicit intent, readable ⇒ pythonic
lst3 = list(lst1)            # constructor call, sometimes seen in the wild; equivalent with:
lst4 = type(lst1)(lst1)      # - type is a built-in, to get the type of an object instance
lst5 = lst1.__class__(lst1)  # - the type reference is stored in the attribute __class__
```

To be explicit, the *lst4* example is interpreted the same as *(type(lst1))(lst1)*

- Regarding object identity, be prepared for surprises:

```
c = 1    # this is a different, new instance of the integer 1, right?
c is a   # True?!
```

Python treats numeric values as singletons. Also floats:

```
d = 2.71828
e = 2.71828
d is e   # True
```

But obviously, instances of different types cannot be the same object.

```
f = 2.0   # float
g = 2     # int
f is g    # False
```

# Language features

## Names and values – *string interning*

- Surprises, vol. 2:

```
s1 = 'foo'
s2 = 'foo'   # let's make another one
s2 is s1     # True?!
```

*In computer science, **string interning** is a method of storing only one copy of each distinct string value, which must be immutable. Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned.* [https://en.wikipedia.org/wiki/String\\_interning](https://en.wikipedia.org/wiki/String_interning)

- <https://stackoverflow.com/questions/15541404/python-string-interning>
- For week 11, <https://stackoverflow.com/questions/8846628/what-exactly-is-a-symbol-in-lisp-scheme>
- Details of behaviour are implementation- and likely also version-specific. E.g. in CPython 3.4.3:

```
s1 = 'foo bar'
s2 = 'foo bar'
s2 is s1   # False?!?!
```

- If desired, a given string can be interned explicitly:

```
from sys import intern
s3 = intern('foo bar')
s4 = intern('foo bar')
s4 is s3   # True
```

# Language features

## Names and values – None

- The special value **None**:
  - Just a value like any other – not so special, really
  - A singleton of type *NoneType*
  - Commonly used as a placeholder for “not specified”, “not applicable” and similar.
    - E.g. can be a good default value for an optional parameter. Much better than an empty list, because in Python **mutable default parameters are dangerous**:

```
def append_a_one(lst=[]): # append int 1 to given list (default empty)
    lst.append(1)
    return lst
```

```
print(append_a_one()) # [1]
print(append_a_one()) # [1, 1]
print(append_a_one()) # [1, 1, 1]
```

- Unintuitive, but makes perfect sense if we consider what Python is doing. The object instance for the default is created *when the **def** is evaluated* – which occurs only once. Hence, better:

```
def append_a_one(lst=None):
    lst = lst or []
    lst.append(1)
    return lst
```

Now each call using the default will return the one-element list [1], as expected.

# Language features

## Names and values – **global**

- The keyword **global** declares that a name refers to a global:

```
x = 3
def f():
    x = 5
    def g():
        global x    # in “g”, “x” refers to the global “x”, from the top-level scope
        print(x)
    print(x)        # this is the “x” local to “f”
    g()
f()
```

- In the same scope:
  - The same name cannot refer to both a formal parameter and global – **SyntaxError**
  - The same name cannot refer to both a local and a global – **SyntaxWarning** (and behaviour that is likely not what was intended)
- This also implies that assigning a new value to “x” inside “g” rebinds the **global** name “x”, in the top-level scope. (Try it.)

# Language features

## Names and values – **nonlocal**

- Similarly, the keyword **nonlocal** instructs Python to look for the given name in the surrounding scopes:

```
x = 5
def f():
    x = 2
    def g():
        nonlocal x
        x = 3  # without nonlocal, this would create a new “x” for “g” only
    print(x)  # 2
    g()
    print(x)  # 3, our “x” has been re-bound by “g”
f()
print(x)     # 5, this is the global “x”
```

- The purpose of **nonlocal** is to be able to re-bind names that live in an outer (but not global) scope, since Python makes no syntactic distinction between defining a new name and re-binding an existing name to a new value. (Contrast the Scheme subfamily of Lisp, with “define” vs. “set!”.)
  - Because **LEGB**, for just reading or accessing **nonlocal** is not needed.

# Language features

## Imperative vs. functional programming

- **Why talk about paradigms?**
  - To answer, first we must ask, “What is programming?”
    - *Codification of imperative knowledge*, i.e. how-to knowledge (SICP, 2<sup>nd</sup> ed., Abelson & Sussman, 1996).
      - Algorithms: how to solve (particular) problems
      - Data structures: how to store data efficiently for different use cases
    - Usually machine-readable (executable), but not central to the idea.
    - Contrast pure mathematics, which is *declarative knowledge* – e.g. that something exists, but (often) no constructive proof to actually compute it.
  - Programming paradigms concern ***ways to structure that knowledge***.
    - Practical implications for understandability, developer efficiency, maintainability, potential for automated analysis, ...

# Language features

## Imperative vs. functional programming

- Roughly speaking, with a trivial example of each style:
- **Imperative**: a sequence of simple steps, often involving mutable state.

```
def sum(lst):    # input: a list of numbers
    s = 0
    for x in lst:
        s = s + x    # mutable state: "s" is updated
    return s
```

- **Functional**: a sequence of function applications, avoiding mutable state; “executable mathematics”.

```
def sum(lst):
    def loop(acc, lst): # each iteration gets a new "acc", so only defs, no mutation
        if not len(lst): # empty input ⇒ finished, return the accumulated result
            return acc
        else:
            first,*rest = lst
            return loop(acc + first, rest) # ← sequence of function applications here
    return loop(0, lst) # start the loop
```

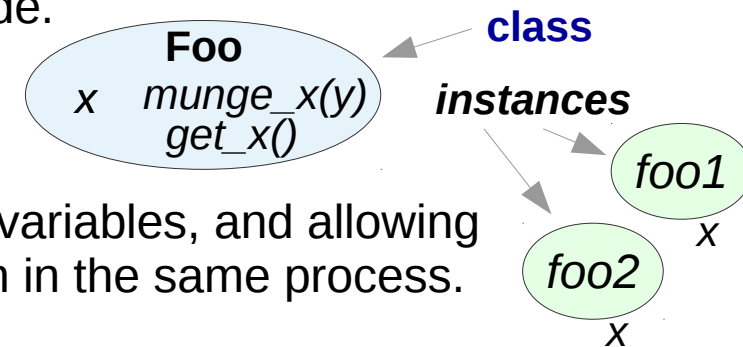
*This pattern is, roughly, what a loop is. (More on this at the end of the course.)*



# Language features

## Imperative vs. functional programming

- Object-oriented programming (OOP) is a subcategory of imperative programming.
- Good at information hiding, and *encapsulation* in the sense of packing data and its related operations into one place in the source code.
- Roughly, object = data + a set of operations on it.
- In scientific computing, good for eliminating global variables, and allowing several independent instances of your solver to run in the same process.
- OOP is more *status quo* than *state of the art*; every programmer should know the basics. Almost all modern languages have some kind of an object system.
- We will discuss the basics of OOP at the end of this lecture.
- Some points on FP:
  - An emphasis on *higher-order functions* (taking functions as arguments), such as **map**, **filter** and **reduce** (a.k.a. **fold**).
  - No mutable state, no side effects. Easier to analyze, and write bug-free code.
  - The main advantage of FP however is an **increase in modularity** (Hughes, 1984). Near the end of the course, we will briefly talk about this.
- A classic textbook explaining both IP and FP is **SICP** (Abelson and Sussman, 1996).



# Lexical vs. dynamic scoping

- In **lexical scoping**, if a variable name's scope is a certain function, then its scope is the **program text of the function definition**: within that text, the variable name exists, and is bound to the variable's value, but outside that text, the variable name does not exist.

By contrast, in **dynamic scoping**, if a variable name's scope is a certain function, then its scope is the **time-period during which the function is executing**: while the function is running, the variable name exists, and is bound to its value, but after the function returns, the variable name does not exist.

[https://en.wikipedia.org/wiki/Scope\\_\(computer\\_science\)#Lexical\\_scoping\\_vs.\\_dynamic\\_scoping](https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scoping_vs._dynamic_scoping)

- Consider nonlocals. Under **lexical scoping**:

```
def f():  
    def g(): # "g" is defined inside "f"  
        print(x)  
    x = 42  
    g()  
f()
```

```
def g(): # "g" is defined outside "f"  
    print(x) # NameError, no "x" in scope  
def f():  
    x = 42  
    g()  
f()
```

- This program prints 42.
- This program crashes.
- Under **dynamic scoping**, the program on the right would also print 42.
  - The modern consensus is that for general use, dynamic scoping spells disaster for maintainability. Hence most languages are lexically scoped.
  - However, convenient in special cases, e.g. passing values (plotter options?) through several layers of functions, without needing to modify what parameters they take.
  - It is possible to emulate dynamic scoping in Python. (More on this in the exercises.)

# Language features

## Strong and duck typing

- **Strong typing** has no single generally accepted definition, but is commonly taken to imply “no implicit type conversions”. This is what we mean by ST on this course.

```
'The answer is ' + 42          # TypeError, cannot + str and int  
'The answer is ' + str(42)    # explicit conversion, OK!
```

- In a real-world program, as mentioned in lecture 2, for this particular use it is better to use string formatting:

```
'The answer is {:d}'.format(42)
```

- **Duck typing** applies the duck test to type safety.
  - “If it walks like a duck and it quacks like a duck, then it must be a duck.”  
[https://en.wikipedia.org/wiki/Duck\\_typing](https://en.wikipedia.org/wiki/Duck_typing)
  - The important point is that the *type* of the value *does not matter*; whatever it is, it only needs to *support whatever we try to do with it*.
  - The presence of the requested attributes and properties is checked at runtime.
  - Similarly, any results from function and method calls are validated at runtime, typically using exceptions to handle any errors.

# Language features

## Duck typing

```
class Duck:
    def quack(self):
        print('quack quack')
```

```
class AustralianSpottedDuck(Duck): # the type in the parentheses is the parent class.
    pass
```

```
class AnimalSimulator3000: # no relation whatsoever to Duck!
    def meow(self):
        print('meow')
    def quack(self):
        print('quack')
```

```
class Dog:
    pass
```

- What happens if we create one of each, and call `x.quack()`, where `x` is the object instance?
  - *Duck* quacks, because it implements *quack* itself.
  - *AustralianSpottedDuck* quacks, because it inherits *quack* from its ancestor *Duck*.
- So far that is just basic OOP (see end of this lecture), but now things get interesting:
  - *AnimalSimulator3000* quacks, **because it implements quack** – a duck typing system, by definition, doesn't care that this object is not related to *Duck*. **This class just happens to implement a method with the expected signature, that does the expected thing.**
  - *Dog* raises an **AttributeError** at runtime, because it does not have a *quack* method. **The compiler does not catch this.**

# Language features

## Call-by-sharing

- In the ***call-by-sharing*** argument passing model, the caller and callee *share the same object instance*:

```
def f(x):  
    print(id(x))
```

```
def main():  
    a = 42  
    print(id(a))  
    f(a)
```

```
main()
```

- Since assignment in Python binds names, the values of any immutable arguments – as the caller sees them after the call – **cannot be changed by the callee**.
  - Exercise: assign a new value to `x` in `f()`, after the print. Add another copy of the print, after the assignment. Also add another copy of the print in `main()`, after the call to `f()`. Observe the result, and explain the behaviour of the program.
- **nonlocal** and **global** cannot help here; they only affect *free variables*, whereas in our “f” in this example, `x` is a formal parameter.
  - A **free variable**, adapting the usual definition to Python's name/value semantics, is a name that is **locally unbound**; i.e. neither a locally defined name nor a formal parameter.

# Language features

## Call-by-sharing

- Since *the object instance is shared*, **mutable arguments can be modified by the callee**, and the caller will see the changes:

```
def g(x):  
    x[0] = 'that'  
  
def main():  
    lst = ['bat', 'man']  
    g(lst)  
    print(lst)
```

```
main()
```

- Mutability is a property of an object instance, not of a name (such as formal param). Hence:
  - The function definition **does not declare** mutability. The function “g” just takes a parameter.
  - The body of this particular “g” expects its argument to be both indexable and mutable, but **it is the responsibility of the caller to honor this contract**. (Duck typing!)
    - In a real-world program, the docstring of “g” should explain what the function expects.
- Using mutable arguments for in-place output (updates) is a good strategy in cases where it makes no sense to make a copy (e.g. large arrays). (Many functions in NumPy have an optional `out=...` argument for this.)
- Keep in mind that *the default value*, if any, usually should not be mutable, due to the details of how Python works (example on slide 20, above).

# Language features

## Call-by-sharing

- How *call-by-sharing* differs from ***call-by-value***:
  - In call-by-value, a new copy of the passed object instance is made, whereas call-by-sharing uses the caller's original object instance.
- How *call-by-sharing* differs from ***call-by-reference***:
  - A reference points to a *storage location*, not (directly) to an object instance.
  - Writing another object instance to that storage location (by assignment to the reference) replaces the original, for both callee and caller.
  - The closest analogue to passing an immutable object is passing a **const** reference (e.g. in C++). However, the constness is a property *of the reference*, not of the object instance.
    - In C, the **pointer** (C's version of reference) and its **target** may have separate constness, although both are properties of the pointer. A const pointer means no reassigning to point it to other things; a const target, no mutation of state. Syntax: **const** T \***const**
    - The const reference gives a language-enforced guarantee that the callee will not modify the referenced object (at least, not through that particular reference; see **aliasing**).
    - See also **design by contract**, which is an approach for design of robust programs.
- Example (C, C++, Fortran 90, Python):  
[https://github.com/Technologicat/python-3-scicomp-intro/tree/master/examples/call\\_by](https://github.com/Technologicat/python-3-scicomp-intro/tree/master/examples/call_by)

# Structuring Python programs

## Magic comments – metadata for OS and Python

- In the real world, useful to start each .py file with these two *magic comments*, which are not shown in the lecture examples (to save space on the slides):

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

- The first line tells the [shell](#) (most often [bash](#)) which program to run this file with.
  - It is used by all \*nix operating systems (such as Linux, Mac OS X), when the user invokes the .py file directly as an executable.
  - This line is called the *shebang* or the *hashbang*, after the `#!` (“hash” and “bang”). (Just like “hashtag”, the name has nothing to do with hashing or hash tables.)
  - The hashbang must be the first line in the file.
  - [/usr/bin/env](#) looks up its argument in the [PATH](#); this is a good way to start the default python3 from the user's current environment, regardless of where it happens to be installed. (Such a lookup is mandatory to correctly support *virtual environments*; see lecture material, p. 66.)
- The second line tells Python which text encoding ([slides 15–16](#)) is used in the file.
  - The default is utf-8, so if the file is saved as utf-8, this is not strictly necessary (but *explicit is better than implicit*).
- See lecture material, pp. 51–52.



# Structuring Python programs

## Strategies

- **Script**, or *“let's quickly throw something together...”*
  - Arbitrary sequence of statements and/or expressions.
  - Good enough for small one-off tasks where any formal organization of code is just “bureaucracy”.
  - Lacks modularity, inconvenient to re-use parts later.
- **Procedural program**
  - Structured into functions (a.k.a. procedures; no difference in Python).
  - Data is stored in passive containers, on which the functions act.
  - Suitable for both imperative and functional programming.
- **Object-oriented program**
  - Structured into **classes**; roughly, active data containers that define the related data and procedures in the same place.
  - Heavy focus on mutable state; almost inherently imperative.
  - Data stored in the object instance is a good way to avoid global variables.

We will look at these modular, maintainable strategies below.

# Procedural program

## Example

```
def double(lst):  # fully FP: runs input through a mapping, constructs a new list
    return [2*x for x in lst]
```

```
def sum(lst):      # hybrid: functional interface to an imperative algorithm
    s = 0
    for x in lst:
        s += x
    return s
```

```
def munge(lst):    # fully IP: mutates its input!
    for k in range(len(lst)):
        lst[k] = lst[k] ** 2
```

```
def main():
    lst = list(range(10))
    munge(lst)
    print(double(lst), sum(lst))
```

```
main()
```

The program is organized into functions.

The idea is that any sufficiently general, common parts can be re-used in and/or by other programs later.

⚠ But don't get too stuck on the ideal of re-use; abstraction and modularization, even just for the program at hand, are already powerful.

...and you can always strip your old programs for parts later.

- Procedural programming is very old; Fortran, ALGOL, COBOL, BASIC ca. 1960.  
[https://en.wikipedia.org/wiki/Procedural\\_programming](https://en.wikipedia.org/wiki/Procedural_programming)

# Procedural program

## Pythonically...

```
"""A procedural program."""
```

```
import math
```

```
def f(x):  
    return x**2
```

```
def main():  
    a = math.pi  
    print(f(a))
```

```
if __name__ == '__main__':  
    main()
```

★ But if you do, consider where the delay occurs when the **import** runs for the first time, and Python actually loads and initializes the imported module.

Also, if the function is called in an inner loop, then perhaps do not import from there, to avoid a performance hit (even if slight) from checking whether the module is already loaded every time the function is called. *Python does not guess.*

- The file docstring must be first (after the magic comments).
  - This is shown if a user does **import** example; help(example)
    - You *can* use semicolons in Python, to separate several statements on the same line.
- Imports are usually placed at the top (after the file docstring), although Python does allow them anywhere. (If you only need something very locally, feel free to **import** inside a function. ★)
- The **if** block at the end is known as the **conditional main** idiom.
  - **Idiom**: language-specific (usually very small) design pattern.
  - This particular idiom allows dual use of the same file, as both a main program and a library.
  - We met `__name__` on slide 5; it contains the current module name.
  - We invoke the function `main()` *only if this file is being executed as the main program*, module name `"__main__"`. When another file **imports** this one, the `main()` defined here will not run.

# Object-oriented program

## Example

"""An OO program."""

```
import math
```

```
class Point:
```

```
    def __init__(self, x, y):
```

```
        self.x = x
```

```
        self.y = y
```

```
    def dist(self, p):
```

```
        return math.sqrt( (self.x - p.x)**2 + (self.y - p.y)**2 )
```

```
def main():
```

```
    p1 = Point(1, 1)
```

```
    p2 = Point(3, 2)
```

```
    print(p1.dist(p2))    # ⇒ 2.23606797749979
```

```
if __name__ == '__main__':
```

```
    main()
```

★ What Python calls *attributes*, other object-oriented languages may call *members* or (in case of data) *fields*.

C++ calls data-containing class attributes *static members*.

- The program is organized into **classes** (in this example, just one).
  - Classes have **attributes**. ★
  - Roughly speaking, an attribute can be data, or a function.
    - Data may be an object instance. See [object composition](#).
  - Functions that are attributes of classes are called **methods**.
- There are *instance attributes* and *class attributes*. **Instance attributes** belong to an object instance, whereas **class attributes** are shared among all instances of the class.

# Object-oriented program

## Technical points

"""An OO program."""

**import** math

Inherits from *object*, by default since nothing specified.

**class** **Point**:

Current object instance; name arbitrary, “*self*” is customary.

**def** **\_\_init\_\_**(*self*, x, y):

*self*.x = x

*self*.y = y

*Self* is the first formal parameter of all *instance methods*.  
Must be declared explicitly, but is passed implicitly.

**def** **dist**(*self*, p):

Instance attribute

return math.sqrt( (*self*.x - p.x)\*\*2 + (*self*.y - p.y)\*\*2 )

In this example, *dist* is an instance method;  
it uses data “from this object instance” (*self*).

**def** **main**():

p1 = Point(1, 1)

p2 = Point(3, 2)

print(p1.dist(p2)) # ⇒ 2.23606797749979

p: the other Point to which to compute  
the (Euclidean) distance.

(To be exact, in duck typing, p is *any object*  
that just happens to have “x” and “y”,  
with compatible data stored in them.)


**if** **\_\_name\_\_** == '**\_\_main\_\_**':  
main()

- In Python, the **constructor** is always named **\_\_init\_\_** (special method; double underscores).
  - The constructor initializes a new object instance, which it then returns (even if no **return**).
  - It is called when the user does `p = Point(1, 2)`
- Note the two different uses of x in the constructor. The bare x is the formal parameter, whereas *self.x* is an instance attribute.
- In Python, data members are **not declared**; just create them by assignment. Typically this is done in the constructor, but Python allows creating new attributes at any time, from anywhere.

# Object-oriented programming

## Data structure with named fields

- This is also conveniently made using **class**:

**class** **MyDataStore**:  In Python, classes are conventionally named in **CamelCase**.  
    `z = 42`      # in Python, **class attributes** are initialized here

**def** **\_\_init\_\_**(*self*, x, y):  
        *self*.x = x    # **instance attributes** are initialized here  
        *self*.y = y

    # no need for other methods

`mds1 = MyDataStore(x=2, y=3)`  
`mds2 = MyDataStore(x=5, y=17)`

`print(MyDataStore.z)`    # 42  
`print(mds1.z)`          # same, lookup finds “z” in the class

`print(mds1.x)`          # 2  
`print(mds2.x)`          # 5  
`print(MyDataStore.x)`    # **AttributeError**, since “x” is only defined for instances.

`mds1.s = 'hello'`      # legal in Python, but usually bad style (if you need to add new  
                          # attributes, do it in a method instead; better encapsulation, modularity)

Language design and convenience aside; in a sense, from the viewpoint of an object instance, the class is indeed an enclosing scope, so LEGB applies here, too. We will talk about this at the end of the course.

- See lecture material, p. 35; and Python documentation, which suggests **an empty class** for this.

# Object-oriented programming

## Polymorphism

- Since Python names are typeless, the Point example does not specify what type of objects *x* and *y* are. Hence, with a minor modification, we can re-use the same Point class to compute elementwise distances in two *arrays* of points:

```
import numpy as np    # We will talk about NumPy in lecture 4.
import ooex           # The "Point" OO example from slide 36, saved as ooex.py.

def main():
    xx1 = np.array([1, 2, 3, 4, 5], dtype=np.float64) # x values for several points
    yy1 = np.array([1, 4, 9, 16, 25], dtype=np.float64)
    xx2 = xx1.copy() # make a copy so we avoid trouble later if we want to modify p2.x
    yy2 = np.array([1, 1, 1, 1, 1], dtype=np.float64)
    p1 = Point(xx1, yy1)
    p2 = Point(xx2, yy2)
    print(p1.dist(p2))

if __name__ == '__main__':
    main()
```

- This is a form of *polymorphism* inherent in duck typing.
- The required modification to the Point example is to **import** numpy **as** np, and replace `math.sqrt` with `np.sqrt` to make the `dist()` method understand arrays. (Exercise: try this.)
- Exercise: explain the reasoning behind the comment on modifying `p2.x`, above.

# Object-oriented programming

## Class methods, static methods

- *Class methods* and *static methods* are often useful in object-oriented programming:

**class** Example:

```
a = 42                # class attribute
```

```
@classmethod
```

```
def answerize(cls, x): # type implicitly passed; can access class attributes  
    return cls.a * x
```

```
@staticmethod
```

```
def do_stuff(x, y):    # no implicitly passed first parameter; just a function  
    return x + y
```

- A **static method** is really just a function that lives in the namespace of the class.
  - Commonly used for utilities, related to the class but needing no access to attributes.
- A **class method**, in Python, is a method that, instead of *self*, gets a reference (conventional name *cls*) to the *type* it belongs to, providing access to *class attributes*.
- Call syntax for both is *classname.methodname*, e.g. Example.answerize(10), Example.do\_stuff(1, 2)



# Interlude

## Decorators

- The definition of class and static methods uses the *decorator* syntax.
  - The `@classmethod` and `@staticmethod` decorators are built-in; no need to import anything.
- In general, a Python **decorator** is a modifier for a function or method definition.
  - A few exist in the standard library:  
<https://stackoverflow.com/questions/7120342/does-python-have-decorators-in-the-standard-library>
  - The Python wiki provides many decorator recipes:  
<https://wiki.python.org/moin/PythonDecoratorLibrary>
  - Some libraries expose functionality through decorators:
    - Numba provides `@numba.jit` to tag a function for JIT compilation.
    - [\*line\\_profiler\*](#) provides `@profile` to tag a function for line-by-line performance profiling.
    - Some of Cython's options can be controlled per-function by using the decorator syntax.
  - Custom decorators can be created. For links to more reading, see the lecture notes, p. 49.
  - No relation to decorator (design pattern); more accurately described as [\*advice\*](#).
  - Decorators are essentially [\*syntactic sugar\*](#):

<code>@deco</code>		<code>def f(x):</code>
<code>def f(x):</code>	is equivalent with	<code>...</code>
<code>...</code>		<code>f = deco(f)</code>

# Object-oriented programming

## Access modifiers

- **Access modifiers** are keywords in object-oriented languages that set the accessibility of classes, methods, and other members. Access modifiers are a specific part of programming language syntax used to facilitate the encapsulation of components.  
[https://en.wikipedia.org/wiki/Access\\_modifiers](https://en.wikipedia.org/wiki/Access_modifiers)
- C++: private, protected, public
- Java: private, protected, package, public
- **Python has no access modifiers.**
  - Philosophy: “[We are all responsible users](#)” –The Hitchhiker's Guide to Python
  - A short, informative read: <http://radek.io/2011/07/21/private-protected-and-public-in-python/>
  - Official tutorial: <https://docs.python.org/3/tutorial/classes.html#private-variables>
  - Summary:
    - A name beginning with a single underscore is a convention that means *private*; this is sufficient to make Python programmers not touch it.
    - A name with **two** leading underscores (and at most one trailing one) triggers [name mangling](#), sometimes useful in class hierarchies; example in the official tutorial.  
Foo.\_\_dostuff → `_Foo__dostuff`
    - *Special methods* (such as `__init__`) have two leading **and** trailing underscores.

# Object-oriented programming

## Operators in OOP

- As an example an operator, consider multiplication. In Python, the syntax

`a * b`

actually means

`a.__mul__(b)`

- If `a.__mul__` does not exist, or fails (**return** *NotImplemented*), **and if “a” and “b” are of different types**, then Python tries a version with reflected (swapped) operands:

`b.__rmul__(a)`

to see if “b” knows how to multiply by “a”.

- E.g. `int` **cannot** know how to multiply by a `Foo`, but if `Foo` knows how to multiply by an `int`...
- Matrix multiplication has its own operator, `__matmul__` (Python 3.5+).
- Some variant of this approach is common in OO languages; e.g. in C++, `a * b`  $\Rightarrow$  `a.operator*(b)`
- All of Python's *special methods* (a.k.a. *magic methods*), with descriptions:  
<https://docs.python.org/3/reference/datamodel.html#special-method-names>
- The standard operators are also provided as run-of-the-mill functions (useful for FP):  
<https://docs.python.org/3/library/operator.html>

# Object-oriented programming

## Inheritance, example

```
class A:  ← “A” inherits from object, by default.
    def hello(self):
        print('Hello from {}'.format(str(type(self))))
```

**class B(A):** ← “B” inherits from “A”.  
**pass** ←

```
def main():
    a = A()
    b = B()
    a.hello()
    b.hello()
```

(To keep the example simple,  
let's not do anything here.)

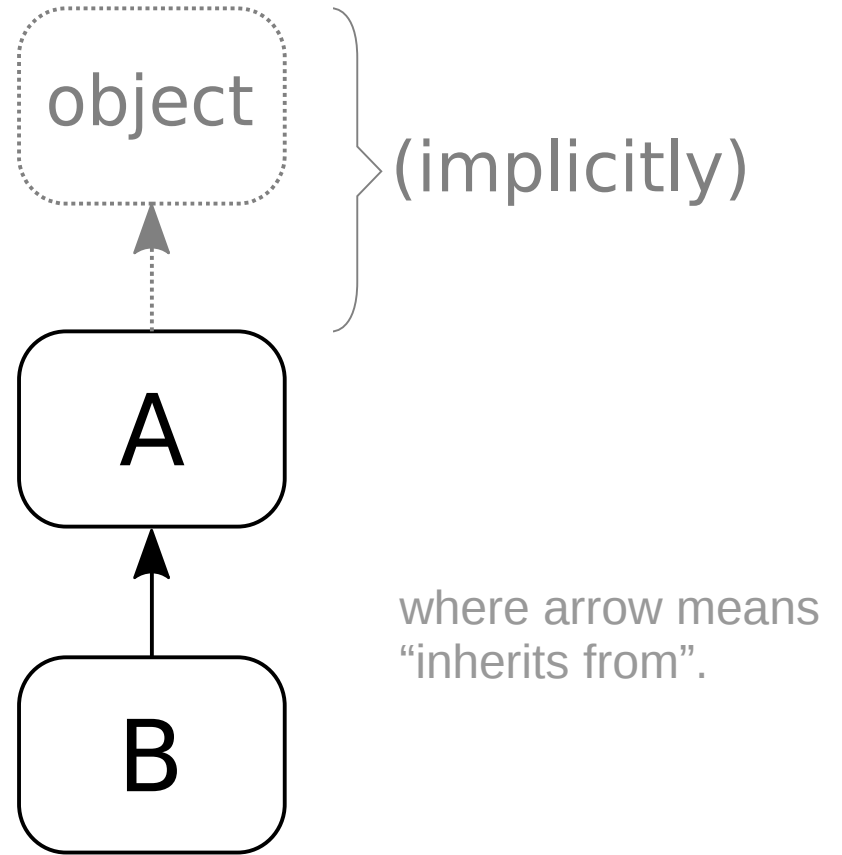
Sometimes actually useful, if the whole point of having a “B” is that its *type* is distinct from that of “A”; e.g. to make custom exception types.)

# main()

But wait, “B” does not define a hello()?

## Terminology – many names for a simple concept:

- “B”: *subclass* of “A”; “A”: *superclass* of “B”
- “B”: *child* of “A”; “A”: *parent* or *immediate ancestor* of “B”  
(object is also **an** *ancestor* of “B”, two links up the chain.)
- “B”: *derived class*; “A”: *base class*



where arrow means  
“inherits from”.

# Object-oriented programming

## Inheritance, example

```
class A:
    def hello(self):
        print('Hello from {s}'.format(str(type(self))))

class B(A):
    pass

def main():
    a = A()
    b = B()
    a.hello()
    b.hello()

main()
```

“A” inherits from *object*, by default.

“B” inherits from “A”.

Python finds `hello()` in the parent class “A”.

# `hello()` is in A, but what is our **dynamic type**?

**Dynamic type** is the actual type of the object instance being handled.

Contrast *static type*, which e.g. in C++ is the type as which the variable (storage location!) holding the object instance has been declared.

- **Inheritance** means that a class is based on another; it will inherit its parent's attributes.
- Roughly, inherited attributes can be used just like attributes defined locally in the class.
  - Since methods are also inherited, inheritance allows code re-use. In this example, “B” does not define a `hello()`, so Python will use the inherited definition from “A”.
- “is-a” relation, specialization (subclass): e.g. *AustralianSpottedDuck* is a subtype of *Duck*.
- Python's **isinstance** honors this relation: `isinstance(b, A) ⇒ True` since “b” **also** is-an “A”.
- [https://en.wikipedia.org/wiki/Inheritance\\_\(object-oriented\\_programming\)](https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming))

# Object-oriented programming

## Inheritance, *overriding*

```
class A:  
    def __init__(self, a):  
        self.a = a
```

```
    def dostuff(self):  
        return self.a * 23
```

```
class B(A):  
    def __init__(self, a, b):  
        super().__init__(a)  
        self.b = b
```

```
    def dostuff(self):  
        return (self.a + self.b) * 42
```

```
foo = B(a=2, b=3)  
print(foo.a, foo.b)  
print(foo.dostuff())
```

In Python, `super()` gets a reference to the “parent part” of `self`.

Somewhat like **nonlocal**, in the sense that it allows accessing names otherwise shadowed by local ones.

Using `super()`, no need to hard-code the parent class name inside methods.

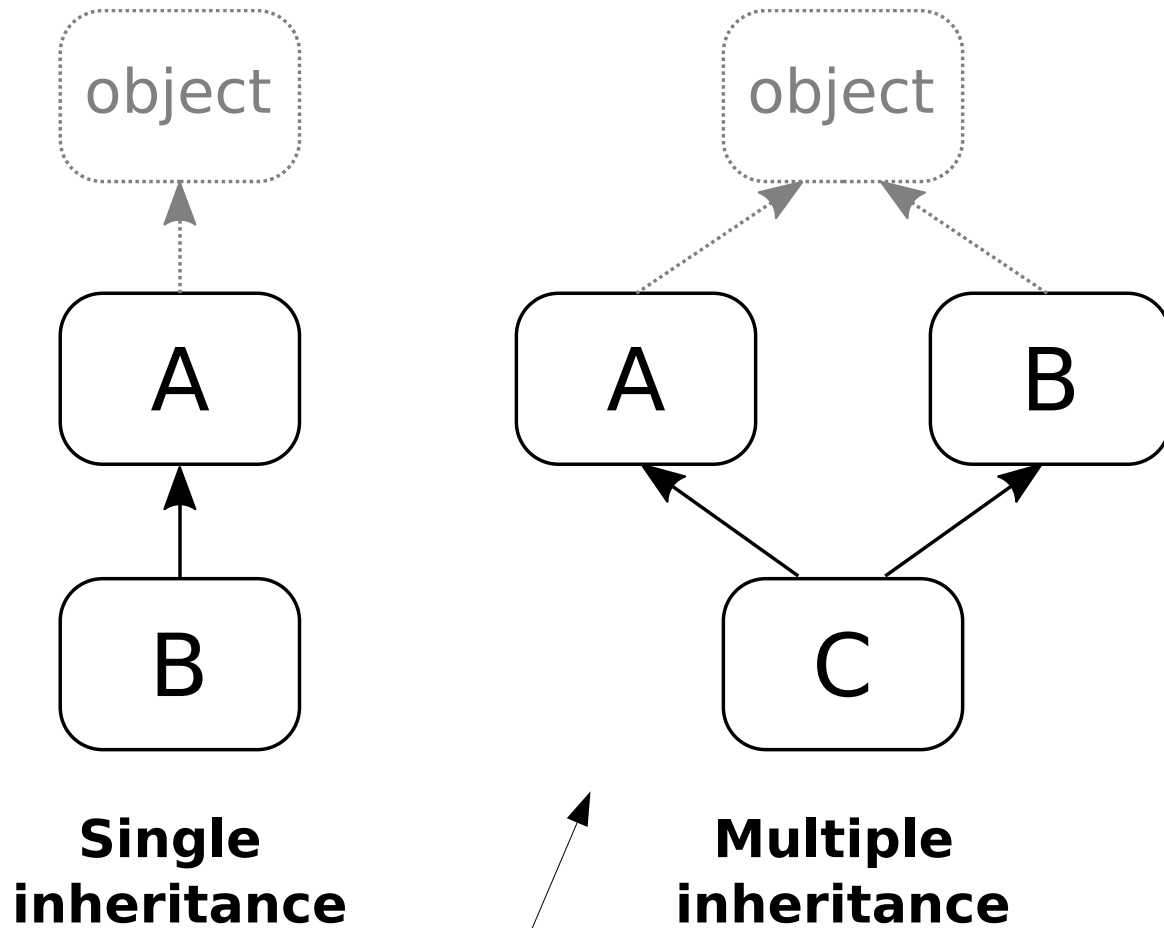
**Overriding** method. For instances of type “B”, this is the `dostuff()` that is used.

Instances of type “A” still use the definition of `dostuff()` in “A”.

- **Overriding** allows specialization of methods by letting each type (class) define its own version of the “same” operation. Python picks which one to use based on the type of the object instance.
- ⚠ Do not confuse with **overloading**, which e.g. in C++ is the practice of *the same class* implementing several methods having the same name, for different input types.
  - Python does not support overloading, but you can `isinstance()` and **if/else** (if you really want to).

# Object-oriented programming

## Inheritance, single vs. multiple



Why would anyone want to do **that**?

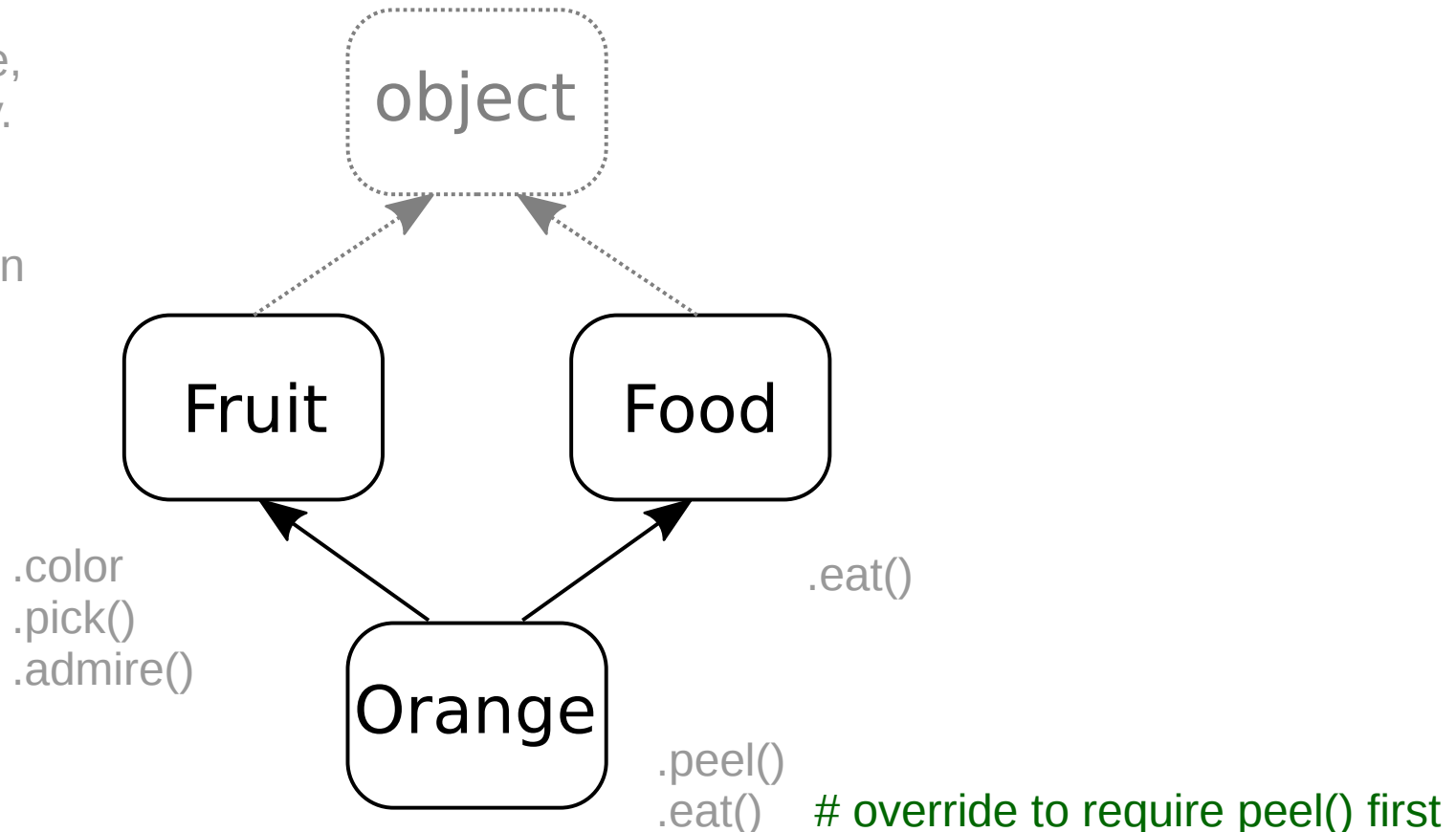
# Object-oriented programming

## Inheritance, multiple

Multiple inheritance is a model for multiple “is-a” relations:

(This is not its only use, but just one possibility.)

Another common use is for **mixins**, to bring in pre-packaged sets of functionality in a piecemeal fashion, without implying an “is-a” relation.)



But how to deal with the diamond shape of the inheritance graph?

- In which order should we search the ancestors for a method implementation?
- What does *super()* even mean in this case?

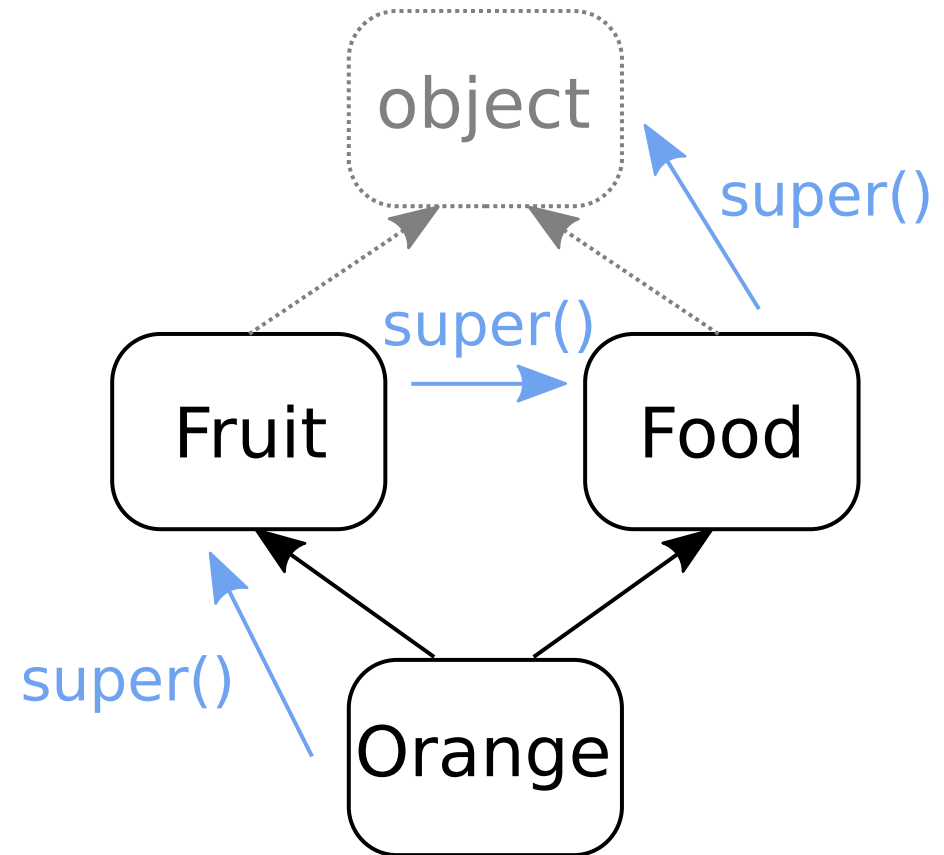


# Object-oriented programming

## Inheritance, multiple

**Method resolution order (MRO)** is determined by an algorithm known as *C3 linearization*.

- The linearization produces a list of ancestors.
  - Assuming `class Orange(Fruit, Food):` , this is `[Orange, Fruit, Food, object]` .
- For a given object instance, Python picks the ancestor list for that type of object.
- When looking up a method, the classes are searched in the order they appear in that list.
  - Hence, lookup order may differ depending on the type of the object.
- `super()` simply picks the next class in the list.
  - `Orange.dostuff()` calls `super().dostuff()`,
    - `Fruit.dostuff()`; if it calls `super().dostuff()`,
    - `Food.dostuff()`, etc.
- Details and a more complex example in Wikipedia:  
[https://en.wikipedia.org/wiki/C3\\_linearization](https://en.wikipedia.org/wiki/C3_linearization)



# Object-oriented programming

## Final words

- *Abstract base classes* are sometimes used to define interfaces.
  - An **abstract base class (ABC)** is a *class* that is meant to be used as a *base* for inheritance only; no instances of it can be created (hence *abstract*).
  - Not so critical in Python, which has duck typing (hence no strict need to formalize interfaces this way), but provided in the standard library module *abc*.
    - In libraries, can sometimes help the library-using programmer visualize the exact kind of duck they need to create.
  - Internally implemented via an advanced feature of Python: custom metaclasses.  
<https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/>
- Python (3.4+) provides an enum type in the *enum module* of the standard library.  
(This also uses a custom metaclass.)
- In short: named constants that behave like integers, but display names when printed.

```
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

- Finally, prefer (object) *composition over inheritance* (a.k.a. *composite reuse principle*):  
[https://en.wikipedia.org/wiki/Composition\\_over\\_inheritance](https://en.wikipedia.org/wiki/Composition_over_inheritance)

# In conclusion

## On abstractions

- Syntax is to programming as notation is to mathematics. If we really, really wanted to, we could:

```
class OOFP:    # object-oriented functional programming? Why not!
    a = 42      # ...and once defined, don't change it (FP style!)
```

```
def __init__(self, b):
    self.b = b    # same remark here: don't change once initialized
```

```
@classmethod
def answerize(cls, x):
    return cls.a * x
```

```
def fp_munge(self, y):    # instance method? In an FP program?
    newb = self.b + y      # Compute something...
    cls = type(self)        # ...find the type of the current instance...
    return cls(b=newb)     # ...and create and return a new instance with updated data!
```

- There are no language-enforced guarantees that no one modifies *OOFP.a* or *someinstance.b*.
  - For immutable object instances with named fields, see *namedtuple* in the [collections module](#).
  - *[advanced]* How to (ab-?)use the context manager to locally enforce no-rebinding:  
[https://github.com/Technologicat/python-3-scicomp-intro/blob/master/examples/no\\_rebind.py](https://github.com/Technologicat/python-3-scicomp-intro/blob/master/examples/no_rebind.py)
    - With bare try/except this would look uglier; the right abstractions can be powerful.
- FP: old state in, new state out. We have rearranged to use *self* instead of an explicit parameter for the old state, with the data stored in the instance attributes. No real gain yet, but see [this](#).

# Meta

## Next time

- Finally getting to the main part of this course:

Overview of, and introduction to, the most important scientific Python libraries!

- See you next week!

