Python 3 for scientific computing

Lecture 2, 31.1.2018
A first practical look at Python 3

Juha Jeronen
juha.jeronen@tut.fi

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

TAMPERE UNIVERSITY OF TECHNOLOGY

Meta

Last week

- Overview
  - What Python is: a general high-level programming solution, open source, ...
  - Why for numerics, why now; how Python compares to MATLAB
  - Language versions: current Python 3 vs. legacy Python 2

- Scientific Python is the bazaar to MATLAB's one-stop shop

- Python in the context of programming languages in general
  - Imperative, somewhat functional, interpreted, object-oriented, duck-typed, ...

Meta

Course schedule

(confidence = 0.9499999999999956)

- Week 1: Introduction and overview.
- **Week 2: A first practical look into Python 3**. Basics by example.
- Week 3: A second practical look into Python 3. Language features by example. Structuring programs in Python.
- Week 4: Overview of scientific Python: NumPy, SciPy, Matplotlib, SymPy.
- Week 5: Basics of NumPy, SciPy, Matplotlib, SymPy, by example.
- Week 6: More NumPy, SciPy, Matplotlib, SymPy. An overview of smaller scientific libraries for specific tasks. Behaviour of floating point numbers.
- Week 7: Parallel computing, in general and in Python. MPI, OpenMPI, mpi4py.
- Week 8: Fundamentals of software engineering: version control (local and social), effective use of comments, testing, static code analysis.
- Week 9: High-performance computing in Python via Cython and OpenMP.
- Week 10: The new paradigm: a gentle introduction to functional programming (FP).
- Week 11: Beyond Python: expressive power, lambda calculus, the Lisp family, and Racket (a modern Lisp).

- **Exercises will be posted after most lectures, starting week 3.**
  - The topics of weeks 1 and 2 are included in the exercises of week 3.
  - One week to solve. Each week starting from week 4, solutions to the previous week's exercises will be discussed.

Meta

General information

- Lecture slides (PDF), uploaded after each lecture:
  https://github.com/Technologicat/python-3-scicomp-intro/tree/master/lecture_slides

- Information on the **final assignment**:
  - A small software project in Python. A simple numerical solver is fine.
  - Individual work. Can be started after week ≈6. Deadline **31.5.**
  - Topic can be freely chosen.
    - Everyone can suggest their own topic. *Is there something that would be useful in your own studies or research?*
    - Default topic, if none suggested:
      - Linear (or bilinear) finite element solver, heat equation, unit square, zero Dirichlet boundary conditions.
      - Use only NumPy, SciPy, Matplotlib et al.; ready-made FEM packages (e.g. SfePy or FeniCS) not allowed.
      - Keep it as simple as possible, this is intended as a small project (and the focus is not on how to make a generic FEM framework).
      - A summary of the required mathematics will be provided.

Getting to know Python (1 of 2)

(This set of slides)

- Python's documentation
- Basic flow control (loops, conditionals) ← From this point on we will have actual code
- ( ) [ ] { } Containers
  - Basic operations on lists and strings ”a” 'a' ”””a”””
  - %0.3g String formatting
  - Slicing (indexing operation)
  - tuple (immutable list)
  - zip (interleaving iterables)
  - Tuple unpacking (destructuring bind and its friends)
  - set, dict
  - List comprehension B = [f(x) for x in A if g(x)]
- 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)

Getting to know Python (2 of 2)

(The next set of slides)

- Modules: defining, loading (**import**)
- The **with** statement (ensuring open files are eventually closed)
- Advanced flow control: *exceptions* (**try**, **except**, **finally**)
- 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)

Python's documentation

- In Python, almost everything is documented

- *help(foo)* to access documentation of object *foo* – such as a function, object instance, or object type.
  - Often contains technical details not easily found by searching the internet ...because if it's in *help*, no need to ask.

- Introspection tools useful to find candidates for *help*:
  - *dir*, *vars* (built-in functions)
  - Standard library module *inspect*, esp. its function *getmembers*

- IPython (incl. Spyder IDE, Jupyter Notebook):
  - *foo? – description*
  - *foo??* – source code (if applicable)

- …

Python's documentation

- ...

- Spyder IDE – *help*, with rich text rendering (NumpyDoc format)
  - Click the object to get help on (to place the text cursor), press *Ctrl+I*
  - If the *Help* pane is not visible, *View* ⊳ *Panes* ⊳ *Help*
  - Depends on static code analysis to determine the type of the object instance under the text cursor – often works, but not always

- Internet!
  - For example, the Python 3 standard library reference: https://docs.python.org/3/library/index.html
  - Project websites (for libraries), e.g. in case of NumPy: https://docs.scipy.org/doc/numpy/
  - Search engines
    - E.g. googling for “python 3 inspect” is likely the fastest way to find this: https://docs.python.org/3/library/inspect.html
    - A useful complement for *help*. Other search results useful when the official documentation is unnecessarily cryptic.
    - See esp. any helpful questions/answers at StackOverflow.
    - Sometimes “python 3” better as a search term than just “python”, because of the still ongoing Python 2 to Python 3 transition.

Basic flow control

Loops, for

- Python supports **for** and **while** loops. Some examples of **for** loops:

```
for k in range(10):  # integers 0, 1, …, 9
    print(k**2)         # ** denotes exponentiation (like in Fortran)

A = [5, 17, 23, 42]   # list, we will discuss them below
for x in A:             # loop directly over a container (here a list)
    print(x)
```

- *Note the consistent indentation*. The loop body **must** be indented.
  - This is how Python knows where the loop body ends in your source code.
- As usual, **continue** skips the rest of one iteration; **break** exits the loop.
- Recall Python's lexical scoping rules from lecture 1: the loop counter lives *in the scope containing the loop* (the loop does **not** introduce a new scope).
  - In other words: the loop counter remains visible after the loop finishes, and has the value it had when the loop finished.
  - Language feature, can be useful.
    https://eli.thegreenplace.net/2015/the-scope-of-index-variables-in-pythons-for-loops/

Basic flow control

Loops, **for**

- Python's **for** loop always runs over an *iterable* (contrast C, Java).

- To iterate over integers, first consider if you really need to.
  - If you only need an index, iterate over the container itself instead.
  - If need to get corresponding elements from two or more sequences, **zip** them (will be discussed later in this lecture).

- If you really do need an integer, use the **range** built-in function, which – roughly speaking – gives an iterable view into the natural numbers.
  - See help(range) for more options (start, step).

  - Range is *lazy*. No array; each number is produced on-demand.
    - This was changed in Python 3.0. In legacy codes (Python 2.x), one may see xrange(), which is essentially equivalent to the Python 3 range().

  - Technically speaking... range() creates a *range object*, which is Python's built-in way to represent ranges. It behaves mostly like a *generator*, except that one can iterate over it multiple times (each time starting from the beginning).

Basic flow control

Loops, **for/else**

- In Python, a **for** loop may have an **else** block:

```
for k in range(10):
    m = k**2
    if m > 100:
        break
    print(m)    # here m ≤ 100; no need for an else block (for the if)
else:           # this else belongs to the for (indentation!)
    print(”All squares were ≤ 100.”)
```

- The **else** in a **for**/**else** means “if no **break** occurred”.

- Although the feature is known as **for**/**else**, mnemonic: (**for**/)**break**/**else**

- Keep in mind you can access the final value of the loop counter also in the else block (this makes **for**/**else** actually useful).

  (But for searching a collection for an item, there is a more pythonic way, see below.)

Basic flow control

Loops, **while**

- Example of a **while** loop:

```
x = 0
while x**2 < 100:
    print(x)
    x = x + 1
```

- As usual, **continue** skips the rest of one iteration; **break** exits the loop.

- Be careful: when writing a **while**, all too easy to forget to update the counter.
  - The **for** loop does that for you, whereas the **while** loop expects you to do that yourself.

- The scope for the loop counter is the same as for the **for** loop, but here it is more explicit (since the counter is initialized manually).

- A **while** loop is not really meant to iterate over a collection; use **for** instead. It is appropriate when the decision on whether to keep looping must be made dynamically, i.e. *while* the loop is running: **while** do_something(…):

Basic flow control

Conditionals, **if** statement

- Python has two forms of **if**. First, the statement form:

```
if 0 < x < 1:    # this works, and evaluates each term (at most) once
    print(”hello, ε!”)
elif x < 0:      # in Python, “else if” is spelled “elif”
    print(”now that's too negative”)
else:
    print(”...”)
```

- As usual, the conditions are tested in sequence until one is true, or the **else** block is reached. The corresponding body runs.

- In Python, most things that are not **None** or **False** are “truthy”. Details: https://docs.python.org/3/library/stdtypes.html#truth-value-testing

- **elif** and **else** parts are optional. There may be several **elif**s.

- Chained comparisons in Python: https://docs.python.org/3/reference/expressions.html#comparisons

Basic flow control

Conditionals, logical operators

- To create more complex conditions, use the logical operators **and**, **or**, **not** (and parenthesize where needed).

- The **and** and **or** operators evaluate one term at a time, from left to right.

- They *short-circuit*, i.e. stop as soon as the result becomes known. Hence:
  - **and** stops evaluation after the first falsey value.
  - **or** stops evaluation after the first truthy value.

- The return value from an expression involving **and** and/or **or** is the *actual value of the last term tested*, not a bool! (No implicit conversions.)

- For decision purposes (also in **if**, **while**), only truthiness matters.

- Examples:

```
a or b              # if a is truthy, return a; else return b
a and b             # if a is truthy, test and return b; else return a
a or b or c         # from the left, return the first one that is truthy, or finally c
a and b and c       # from the left, return the first one that is falsey, or finally c
```

Basic flow control

Conditionals, **if** expression

- Python also has an expression form of **if**, sometimes used in assignments:

  x = some_expr **if** condition **else** other_expr

  In this variant, other_expr is mandatory, because an expression
  must always return a value, and the **elif** part is not supported.

  Similar to C's ternary operator a?b:c, but different ordering: b **if** a **else** c

  History for the curious:
  https://docs.python.org/2.5/whatsnew/pep-308.html
  https://www.python.org/dev/peps/pep-0308/ Conditional expressions (2.5+)

- Also sometimes seen: a conditional assignment using only logical operators:

  x = some_expr **or** other_expr

  This is useful if some_expr may be falsey, and in that case one wants to use
  other_expr (e.g. default value for an optional argument initialized to **None**).

  Equivalent to:

  x = some_expr **if** some_expr **else** other_expr

Containers

Basic operations on lists and strings

- Indexing uses square brackets: [ ]
- **Indices always start from 0**, also in NumPy (contrast Fortran, MATLAB)
- Negative indices allowed: -1 is the last element, -2 second to last, …
- Accesses are bounds-checked (like in MATLAB; contrast C); any access past the end raises IndexError; silent memory corruption cannot occur

- A **list** is a sequence of arbitrary elements
- Lists are represented using square brackets, e.g. ['foo', 'bar', 'baz']

- A **string** is a sequence of characters (a character is a length-1 string)
- Python 3 strings support Unicode *transparently*, e.g. '∂²u/∂x²', '∃x', 'λ', ' あ '
  - i.e. requiring no additional effort from the programmer
- The type name is *str*; e.g. str(1) → ”1”
- String literals are quoted, as usual. Supported styles:
  - Single quote (apostrophe): 'a'
  - Double quote: ”a”
  - Three double quotes: ”””a”””
    - This variant allows embedded newlines. Especially useful for docstrings.
  - Same meaning for 'a', ”a” and ”””a””” (contrast *nix shells, e.g. *bash*)

Containers

Basic operations on lists and strings

- Examples:

```
empty_list = []    # just square brackets [ ]

letters = ['a', 'b', 'c']
numbers = [1, 2, 3]
all = letters + numbers               # for lists, + concatenates
numbers_thrice = 3 * numbers  # for lists, * repeats
```

- Python lists are *heterogeneous*, i.e. elements can have different types:

```
stuff = ['cat', 42, True]
```

- Useful methods: append, extend, insert, pop, remove, sort, reverse

- Strings also support joining via str.join. This is invoked as a method on a str instance that represents the separator:

```
s = ', '.join(letters)                # ⇒ 'a, b, c'
```

- Inverse operation: str.split

Containers

Basic operations on lists and strings

- **Escape sequences**:

  ”This string contains a quote \” and\na newline.”   # \” ⇒ ”, \n ⇒ newline

  Full list:
  https://docs.python.org/3/reference/lexical_analysis.html#string-and-bytes-literals

  General explanation:
  https://en.wikipedia.org/wiki/Escape_sequence#Programming_languages

- Specifically for quotes, can also use alternate styles (also ”””...”””):

  ”I'm a string containing an apostrophe.”
  'The cat said ”meow”, so this string contains two double-quotes.'

- **Raw strings** disable escape sequences (Python-specific feature):

  r”Here \n is just backslash and n.”   # prefix r denotes a raw string

  (Very convenient with LaTeX math in Matplotlib labels, and with regexes.)

Containers

String formatting

- A.k.a. *string literal interpolation*. Or in plain English, *insertion of values into a string at runtime, based on a template with placeholders*. Critically important for (human-language) localization; also makes code look clearer than concatenating pieces via +. Currently, three ways to do it:

- **Traditional**, also known as C sprintf style. In Python, % interpolates strings:

```
msg = "Hello %s, the result is %0.6g." % ("Fred", 17/23)
```

  https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

- Python 3.0 and later, **str.format**, invoked as a method of a str instance that is the template:

```
msg = "Hello {name:s}, the result is {result:0.6g}.".format(name='Fred', result=17/23)
```

  https://docs.python.org/3/library/string.html#formatspec

- Python 3.6 and later, **f-strings**, i.e. *formatted string literals*. Can use Python expressions inline.

```
name = "Fred"
result = 17/23
msg = f"Hello {name:s}, the result is {result:0.6g}."  # prefix f denotes a formatted string
```

  https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals

- Examples in plain English: https://pyformat.info/

Containers

Slicing (indexing operation)

- Like MATLAB, Python supports *slicing*:

```
s = 'category'
s[:3]              # ⇒ 'cat'
```

- Full syntax: *start*:*stop*:*step*
  - **CAUTION**: First *stop*, then *step*, unlike MATLAB!
  - Slicing only allowed in indexing expressions
    - See np.r_ if you absolutely must 0:101:10 in MATLAB style, but prefer np.linspace(0,100,11) for that.
  - All parts optional; *start* is inclusive; *stop* is one-past-end; default step is +1
  - For step > 0, default start is 0, default end end-of-string
  - For step < 0, default start is -1, default end start-of-string

- Examples:

```
s[4:6]   # elements 4, 5 (counting from 0!)
s[1:]    # all elements, starting from element 1 (the second one)
s[:-1]   # all elements except the last one
s[1::2]  # starting from element 1, every other element
```

Containers

Tuple (immutable list) vs. list

- **Lists** are *mutable*, i.e. elements may be added or removed:

  ```
  letters = ['a', 'b', 'c']
  letters.append('d')         # add a new single element to the end
  letters.extend(['e', 'f'])  # concatenate another list to this one, modify in-place
  print(len(letters))         # how many letters now?
  ```

- **Tuples** are *immutable*, i.e. once the tuple is created, elements cannot be added or removed. Useful because:
  - Immutable objects are often *hashable*, i.e. can be used as dictionary keys and be included into (mathematical) sets.
  - Language-enforced *guarantee* that there will be no modifications.

  ```
  letters = ('a', 'b', 'c')   # parentheses denote a tuple
  just_one = ('a',)           # one-element tuple (must use a trailing comma
                              #    to distinguish from parenthesized expression)
  the_empty_tuple = ()        # the empty tuple is a singleton (i.e. there is only one)
  ```

- However, keep in mind that the container type says nothing of whether each element itself is mutable or immutable.

Containers

zip (interleaving iterables)

- Useful for obtaining the corresponding elements from two iterables (e.g. lists or tuples) of the same length. Example:

```
T1 = (1, 2, 3)
T2 = ('a', 'b', 'c')
print(tuple(zip(T1,T2)))  # ⇒ ((1, 'a'), (2, 'b'), (3, 'c'))
```

- Technical detail: we need the final tuple() because Python 3 treats some sequences lazily; often that is very useful. (The zipped sequence is generated on-demand rather than all-at-once; tuple() forces the generator.)

- Zipping is often used in loops:

```
for a,b in zip(T1,T2):
    print("Our values are %s and %s" % (a, b))
```

- Note the use of two counter variables, which step in sync; Python *unpacks* each element of the sequence given on the right-hand side of **in**.

- Mnemonic: think of a zipper, the physical object.

Containers

Tuple unpacking

- Regarding the loop with zip(), we mentioned *tuple unpacking*. Examples:

```
a,b,c = (0,1,2)             # ⇒ a = 0, b = 1, c = 2
a,b,*others = range(5)      # ⇒ a = 0, b = 1, others = [2, 3, 4]
*rest,last = range(5)       # ⇒ rest = [0, 1, 2, 3], last = 4
```

- The operation *de-structures* the tuple (or any sequence) on the right-hand side, and *binds* parts of it to names on the left-hand side. Hence *destructuring bind*.

- The star means “the rest”. In an assignment, there can be only one star, and up to Python 3.4, only on the LHS. Python 3.5+ allows some uses of a star on the RHS:

```
T1 = range(5)
T2 = [100, *T1, 200]   # ⇒ T2 = [100, 0, 1, 2, 3, 4, 200]
```

  https://www.python.org/dev/peps/pep-0448/ (Additional unpacking generalizations)

- https://stackoverflow.com/questions/2238355/what-is-the-pythonic-way-to-unpack-tuples
  https://stackoverflow.com/questions/2921847/what-does-the-star-operator-mean

- https://www.python.org/dev/peps/pep-3132/ (tuple unpacking in Python 3.0+)

- See also lecture material, p. 36.

Containers

Unordered containers, **set**

- A Python **set** represents a mathematical set: an *unordered collection of unique elements*. Elements must be *hashable*, which implies immutable. Elements can have any type, and can be heterogeneous.

```
S = {'a', 'b', 'c', 'd'}
S.add('e')   # add an element to the set; if duplicate, discarded automatically
S.remove('a')

empty_set = set()  # no “the”, mutable. Syntax {} means empty dict instead (historical reasons).
```

- Other interesting methods: intersection, union, difference

- Searching a collection for an item, pythonic way. Use the **in** operator:

```
if 'b' in S:   # for the opposite, use “not in”:   if 'b' not in S:
    print(”Yes, 'b' is in S”)
else:
    print(”No, 'b' is not in S”)
```

- **Sets** themselves are mutable, **frozensets** immutable (useful as dict keys).

```
F1 = frozenset(('a', 'b', 'c', 'd'))    # make tuple, convert to frozenset
F2 = frozenset(S)                        # convert a set to frozenset
```

- Note double parentheses for F1; the frozenset constructor wants just one iterable.

Containers

Unordered containers, **dict**

- A **dict** (*dictionary*, a.k.a. *associative array*, *map (data structure))* is an *unordered collection of key-value pairs*. Keys must be *hashable*, which implies immutable. No restrictions on values. Keys and values can have any type, and can be heterogeneous.

- Dictionaries are a highly useful as small ad hoc data structures, since the keys can be human-readable strings.

- To read or write a **dict**, index it with the desired key.

```
D = {'a': 1, 'b': 2, 'c': 3}
D['a']   # ⇒ 1
D['d'] = 4

if 'b' in D:  # search the keys for 'b'
    print(”Yes, 'b' is in D, the corresponding value is %s” % (D['b']))
else:
    print(”No, 'b' is not in D”)

empty_dict = {}   # no “the”, because dict (just like set) is mutable.
```

Containers

Unordered containers, **dict**

- How to iterate over a **dict**:

```
for k in D:              # iterating over the dict itself iterates over the keys
    print(k, D[k])

for k in D.keys():       # same; old way, still sometimes seen in the wild
    print(k, D[k])

for v in D.values():     # values only, no access to corresponding keys
    print(v)             # because dict is one-way

for k,v in D.items():    # useful when you need both keys and values
    print(k, v)
```

- Read from a **dict**, with a default value if the key is not there:

```
D.get('some_nonexistent_key', 42)
```

(Indexing the dict with a nonexistent key instead raises KeyError.)

Containers

Unordered containers, **dict**

- Some final points on **dict**:
  - Keys are (operationally) a set, i.e. an *unordered* collection.
  - To display a sorted list of keys (e.g. for debugging), use

    ```
    print(sorted(D.keys()))
    ```

    Sorted is a built-in function in Python, which takes an iterable and returns a sorted copy. See also **reversed**.

  - To update a dict using the contents of another, use .update:

    ```
    D = {'a': 1, 'b': 2, 'c': 3}
    E = {'d': 4, 'e': 5, 'f': 6, 'a': 100}
    D.update(E)    # insert/overwrite into D, using data from E
    ```

    Any new keys will be added, and existing keys will overwrite.

  - If, at some point, you happen to need a dictionary that remembers the order the items were inserted into it, see the standard library module **collections**, and there **OrderedDict**.

Containers

List comprehension

- *List comprehension* is a syntax for building new sequences from existing sequences by filtering and mapping. In Python:

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

  says that:

  - **for** x **in** A: Consider the elements x of the sequence A, in order.
  - [**filter step**] Accept those elements for which g(x) is truthy.
  - [**map step**] Pass each accepted x to the function f, and record its result.
  - Build the output sequence from these results.
  - Bind the output sequence to the name B.

- The notation used in Python borrows ideas from set-builder notation and the axiom schema of comprehension, from set theory in mathematics.
  https://en.wikipedia.org/wiki/Set-builder_notation
  https://en.wikipedia.org/wiki/Axiom_schema_of_specification#Unrestricted_comprehension

Containers

List comprehension

- The filter step (**if** …) is optional. Similarly, the map step may also just pass through its input, as in [x **for** x **in** range(10) **if** g(x)].

- Using round parentheses in place of square brackets makes instead a *generator expression*, which evaluates its output lazily (on-demand).

- Python has also **set comprehension** and **dict comprehension**.
  The syntax is similar to list comprehension, but with curly braces.

- **Set comprehension**:

  ```
  evens = {x for x in range(10) if x % 2 == 0}   # for numbers, % is “modulo”
  ```

- **Dict comprehension**:

  ```
  D = {x: x**2 for x in range(100)}  # map x ↦ x² for x ∈ [0, 100)
  ```

  Note the colon in the expression part, separating the key and value.
  This is the only syntactic difference between a set comprehension
  and a dict comprehension in Python.

Functions

defining; return values

- To define a function, use the keyword **def**:

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

- Preferred naming convention is *all_lowercase_with_underscores*.
  For mathematical functions just *f* or similar is fine.

- No distinction between functions and procedures (contrast Fortran).
  - Any function may (but does not have to) return a value.
  - If no **return** statement, the return value is the special value **None**.
  - The return value is not declared, and is not part of the function signature.
    Just “**return** some_expr” if you want to return a value.
    - But if you really like statically typed languages, see:
      https://www.python.org/dev/peps/pep-0484/ Type hints (3.5+)
      http://mypy-lang.org/ Mypy, optional static type checker for Python

- The return value can be any Python object, including containers.
  - Output arguments needed very rarely (contrast Fortran).
    But if needed, can use mutable objects as arguments, for in-place output of e.g. large arrays.

- To return several objects at once, pack them into a container such as tuple:

```
def my_other_function(a, b):
    return (2*a, 3*b**2)
```

Functions

Defining, anywhere

- Functions can be defined anywhere, also inside other functions. Arbitrarily deep nesting is allowed. In this example, we have just two levels for clarity:

```
def f(x):
    def g(y):
        return (2*y, 3*y)
    return g(x)
```

- An inner function is inside the outer function's lexical scope, hence it can access also any names visible in the outer function:

```
def f(x):
    def g(y):
        print('a is {0:d}, x is {1:d}, y is {2:d}'.format(a, x, y))
    a = 23
    g(y=42)  # the argument of g is always y; we're just being explicit
```

- **Note!** “a” does not yet have a value when Python reads the definition of g(y).
    - That's fine, because we are just defining “g” (not running it yet), and the name “a” is visible in the whole body of “f” (the lexical scope in which “a” is created).

- Only attempting to actually use “a” before setting a value will raise NameError.

Functions

Argument passing: named args

- Function arguments *can be passed by name* as well as by position.
  (This is one of Python's great strengths for improving readability.)

```
def f(a, b):        # Here "a" and "b" are named.
    print('a = {:s}'.format(a))
    print('b = {:s}'.format(b))

f(a=2, b=3)  # pass both "a" and "b" by name
f(b=3, a=2)  # when using names, ordering does not matter
f(2, b=3)    # pass first argument by position, "b" by name
f(2, 3)      # pass both by position
```

- Tuple unpacking, *, is allowed in function calls. It passes by position:

```
lst = (2, 3)
f(*lst)      # unpack lst into positional arguments of "f"
```

- ...as well as is **dictionary unpacking**, **. It passes by name:

```
dic = {'a': 2, 'b': 3}
f(**dic)     # keys must be arguments accepted by "f", by name
```

Functions

Terminology: *formal parameters*, *arguments*

```
def f(a, b):
    ...
```

Formal parameters: placeholder names, to be filled at call time with...

```
f(2, 3)
```

Arguments: ...actual values (object instances)

```
x = 5
f(a=x, b=10)
```

This call to “f” sets the formal parameter “a” to the current value of “x” (*passes in* the object “x” currently points to in the caller's scope), and sets the formal parameter “b” to 10 (by internally creating a temporary object).

```
a = 3
f(a=a, b=10)
```

This call sets f's “a” to the current value of *the caller's* “a” (and “b” to 10).

Functions

Terminology: *shadowing/masking*

```python
# names defined at the module's top-level scope
a = 1
x = 2

def f(a, b):
    print(a)   # 3
    print(b)   # 5
    print(x)   # 2

f(3, 5)
```

The formal parameter “a” ***shadows*** (hides) the “a” defined in the top-level scope.

- *In computer programming, **variable shadowing** occurs when a variable declared within a certain scope (decision block, method, or inner class) has the same name as a variable declared in an outer scope. At the level of identifiers (names, rather than variables), this is known as **name masking**. This outer variable is said to be **shadowed by** the inner variable, while the inner identifier is said to **mask** the outer identifier.* https://en.wikipedia.org/wiki/Variable_shadowing

- Recall Python's **LEGB** rule for name lookup (lecture 1, slide 24).

Functions

Argument passing: collecting by *, **

- In the list of formal parameters, * makes the function accept any number of positional arguments, collecting any extra ones into a list. In the function body, this list is bound to the name immediately following the *:

```
def f(*args):
    for x in args:
        print(x)

f(2, 3, 5)
```

- Here “extra” means leftovers after positionally passed values have been assigned to all named parameters in the function definition. (Example on next slide.)
  - To keep this short, we will omit discussion of corner cases.

- Similarly, ** makes the function accept any named arguments, and collects into a dictionary any that do not match the name of any named parameter in the function definition. In the function body, this dictionary is bound to the name immediately following the **:

```
def f(**kwargs):    # kwargs is a common abbreviation for “keyword arguments”
    for k,v in kwargs:
        print('{:s} → {:s}'.format(k, v))

f(a=2, b=3, c=5)
```

Functions

Argument passing: example

- The different argument passing styles may be mixed freely:

```
def f(a, b, *args, **kwargs):
    print('a = {:s}'.format(a))
    print('b = {:s}'.format(b))
    for x in args:
        print(x)
    for k,v in kwargs:
        print('{:s} → {:s}'.format(k, v))
    if 'foo' in kwargs:    # check for given dictionary key
        print('foo given, its value is {:s}'.format(kwargs['foo']))

f(a=2, b=3, c=5)  # a and b by name; no formal param c, so c into kwargs
f(2, 3, c=5)      # first two by position, c into kwargs
f(2, 3, 5)        # all by position; since f has just two named formal
                  # parameters, the third positional arg goes into args
```

- The names “args” and “kwargs” are conventional; any name can be used.
  - Sometimes seen instead of “kwargs”: “kwds”, “kws”, “kw”

Functions

Argument passing: optional args

- Pythonic way to implement optional arguments:

```
def f(a, b, c=42):    # here c is optional; if not given, default to 42
    print('a = {:s}'.format(a))
    print('b = {:s}'.format(b))
    print('c = {:s}'.format(c))

f(2, 3)           # use default value for c
f(2, 3, 5)        # pass in a value also for c, overriding the default
f(2, 3, c=5)      # clearer style, especially if “f” supports more than one optional arg
```

- Another way, for rarely used optional arguments:

```
def f(a, b, **kwargs):
    c = kwargs['c'] if 'c' in kwargs else 42
    print('a = {:s}'.format(a))
    print('b = {:s}'.format(b))
    print('c = {:s}'.format(c))
```

- This way, rarely used optional args won't clutter the function signature in *help*
- On the other hand, must document manually so that users know what optional arguments exist. (No guarantees on which keys “f” reads from kwargs!)

Functions

Argument passing: *keyword-only* args

- Finally, Python 3 allows defining also *keyword-only arguments*:

  ```
  def f(*, a, b):
      print('a = {:s}'.format(a))
      print('b = {:s}'.format(b))
  ```

- A lone star in the formal parameter list means that all the parameters that follow it *can be passed by name **only***:

  ```
  f(a=2, b=3)  # pass both “a” and “b” by name
  f(2, 3)      # TypeError, “f” takes no positional arguments
  ```

- Useful to improve readability, to protect against passing arguments by position for functions for which it could be confusing (e.g. if the arguments naturally form a set, instead of an ordered sequence).

- https://www.python.org/dev/peps/pep-3102/ Keyword-only arguments (3.0+)

Functions

Argument default values

- To set default values for optional arguments, use the assignment notation (like in C++):

```
def f(a=42):
    …
```

- If the exact value of the default is an implementation detail, use **None** to signify “not given”, and set the actual value inside the function body:

```
def f(a=None):
    a = a or 42
    …
```

(Otherwise someone will eventually pass in 42 just to explicitly signify “yes, I thought about it, and I want to use the default here”.)

- This follows the programming principle of *information hiding*:
  - Protects the user from unimportant internal details.
  - Call sites do not need changes even if those details change later.

Functions
Documenting

- **Docstrings** (*documentation strings*) are where all the *help* comes from.
- In Python, it is just the first expression in a function body, if that first expression is a string:

```
def rectangle_area(a, b):
    """Compute the area of a rectangle.

Parameters:
    a: number
        The width.
    b: number
        The height.

Returns:
    Number:
        The area."""
    return a*b
```

- Note indentation. The start-quote is indented, but the following lines (inside the docstring) do not have to be. The three-double-quotes syntax is very useful here.
- Spyder's Help window supports pretty-rendering the format used here.
- Docstrings are easy to make. Highly useful when you return to your old code later.

Functions

as first-class objects – as arguments

- In Python, functions are *first-class objects*. They can be:
  - passed in as arguments to other functions
    ⇒ *higher-order functions*
  - returned from functions
  - saved to variables (names), stored in containers, …

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

def do_stuff(op, lst):  # "op" short for "operation"
    result = []
    for x in lst:
        result.append(op(x))
    return result

L = range(10)
K = do_stuff(op=f, lst=L)
```

This is an executable pseudocode sketch of the higher-order function commonly known as **map**.

(Try it, it works.)

It has nothing to do with the data structure also called **map** (dict, associative array).

(Except that both are instances of the mathematical idea of mapping.)

Functions

as first-class objects – as return values

- The *function factory* is a design pattern where we return a function instance.
  It can be used to parameterize operations that have a common template, such as:

```
def make_adder(inc):
    def adder(x):
        return x + inc
    return adder

f = make_adder(inc=3)    # create function from template, bind to name "f"
print(f(2))   # 5
```

- This particular use of a function factory utilizes *lexical closures*.
  - Roughly, a *closure* is a function together with an *environment* that contains the values for its *free variables*. *Lexical* due to lexical scoping.
  - We will return to this in week 9, on functional programming.
  - https://en.wikipedia.org/wiki/Closure_(computer_programming)

- In short, in each adder instance created here, "inc" is bound to the object the caller gave when calling make_adder to create that particular instance – the created function instance "remembers" the value (object reference, to be exact).

Functions

as first-class objects – **lambda**

- Python supports creating short *anonymous functions*, using the keyword **lambda**. These are useful as throwaways for one-off tasks, saving the verbosity of a **def**:

  ```
  lst = range(10)
  result = tuple(map(lambda x: x**2, lst))
  print(result)
  ```

  Python provides **map** (the higher-order function) as a built-in.

  ...but for this particular application, prefer a list comprehension (more readable).

- The body must consist of a single expression only. No statements allowed.
  - Ways around: use a tuple as a container, and index to select the return value. Abuse the short-circuiting logical operators to conditionally sequence tasks.
    - But this is seriously frowned upon, for good reason: such approaches significantly harm readability.
  - Generally, the Python community prefers using **def** almost everywhere.

- Beware: since **lambda** is a keyword, “lambda” is not available as a variable name.
  - Many numerical Python programmers use “lamda” or “lam”. Can also use “λ” (U+3bb, GREEK SMALL LETTER LAMDA), but perhaps not recommended. ftp://ftp.unicode.org/Public/UNIDATA/UnicodeData.txt

- Python's **map** returns a lazy sequence, hence we call tuple() to force the result.

- Python's **lambda** x: … is like MATLAB's @(x) ...

Spyder IDE

Getting started

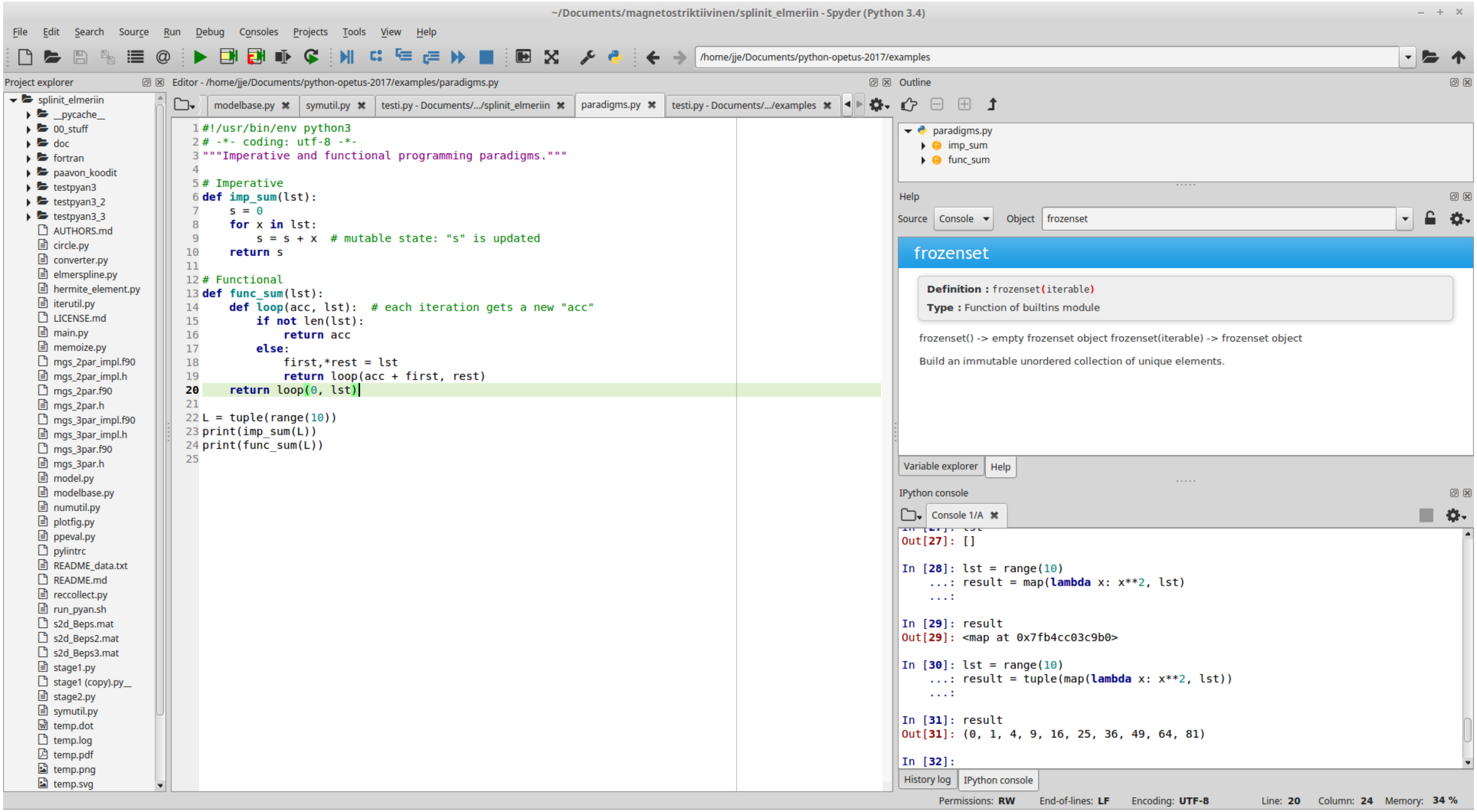
- To start: *All programs* ⊳ *Anaconda* ⊳ *Spyder*
- May be named “Spyder3” for the Python 3 version

(If this appears, things are going well.)

Spyder IDE

GUI layout

- Once loading completes, Spyder looks approximately like this:

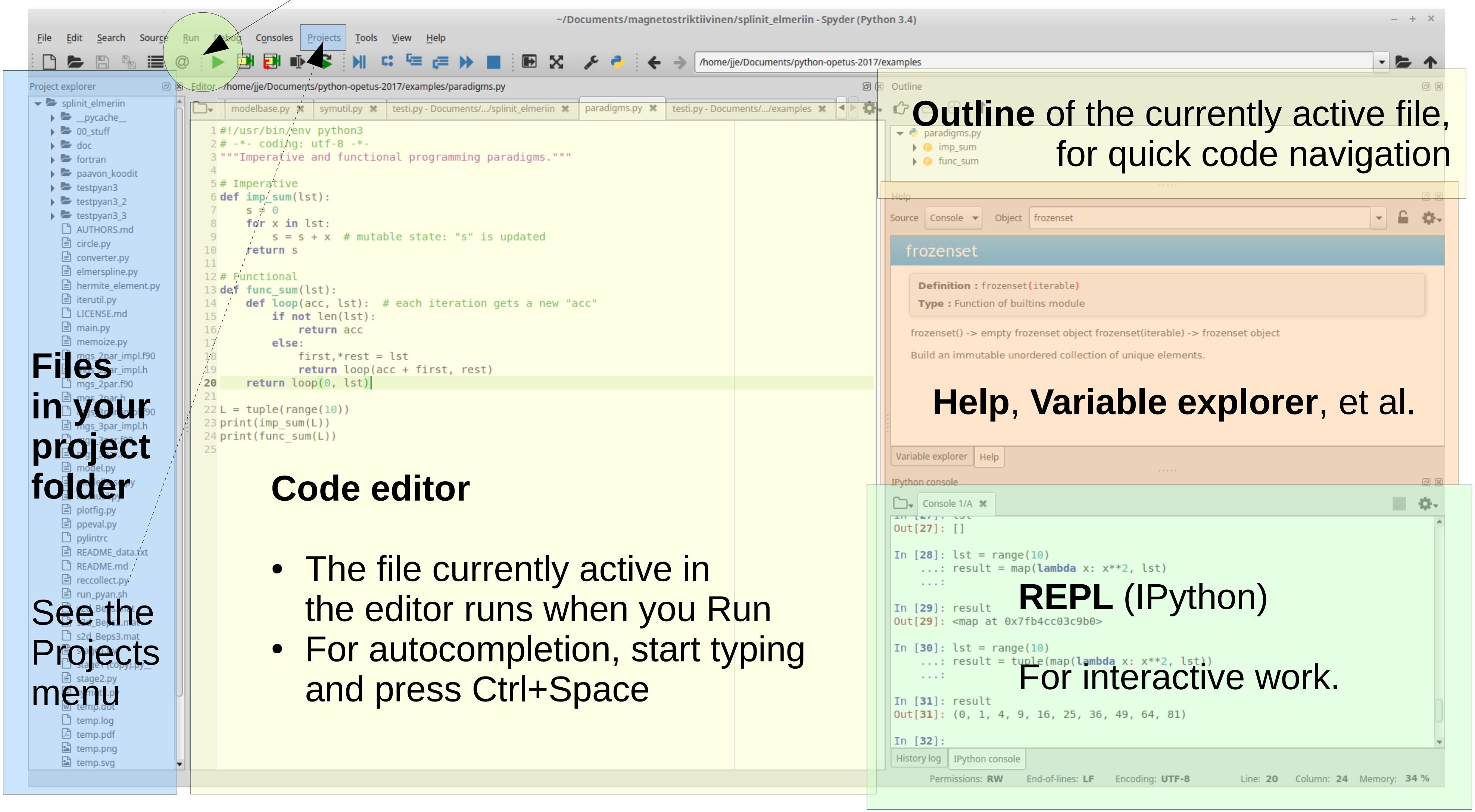


Spyder IDE

GUI layout

Run menu and button "▷"

Can also run by pressing F5.



Files in your project folder

See the Projects menu

Outline of the currently active file, for quick code navigation

Help, **Variable explorer**, et al.

Code editor

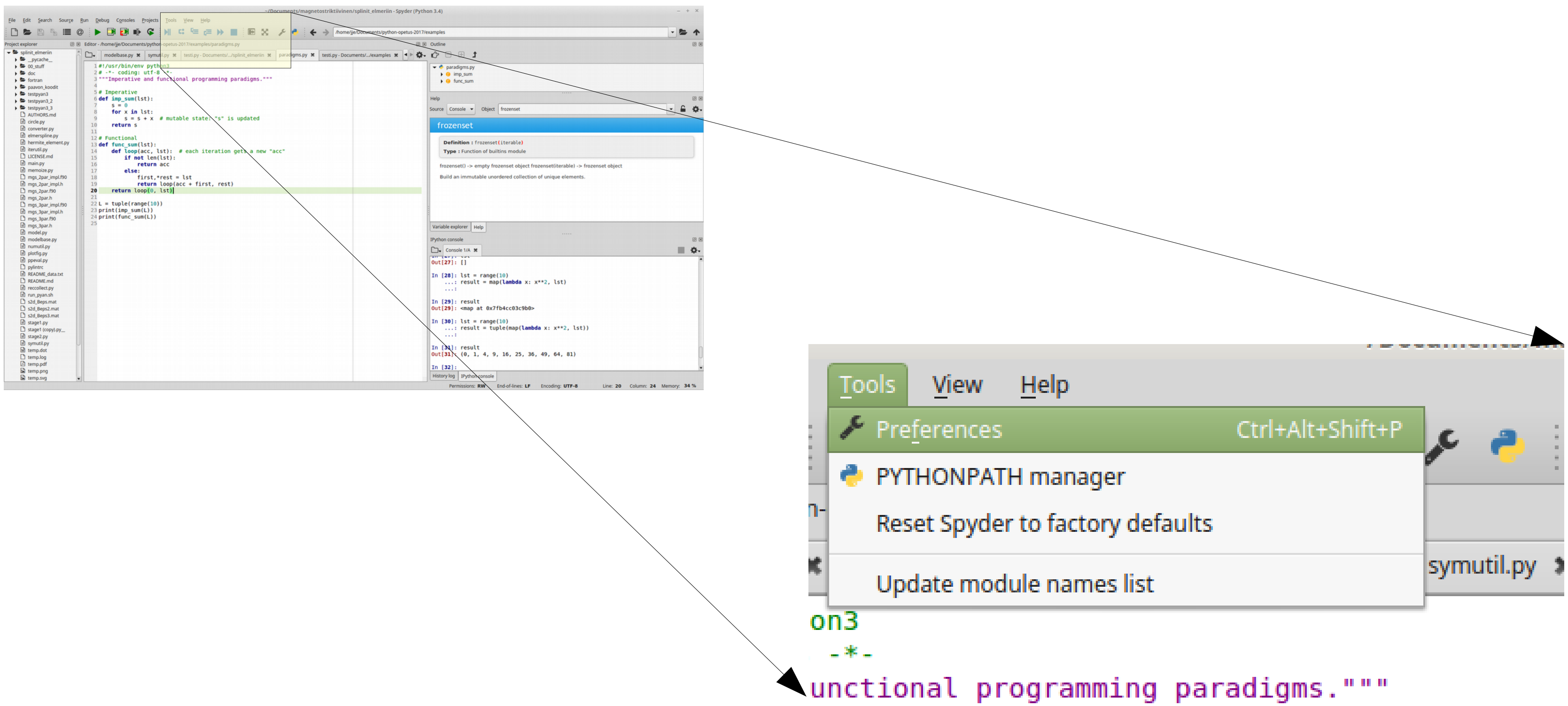
- The file currently active in the editor runs when you Run
- For autocompletion, start typing and press Ctrl+Space

REPL (IPython)

For interactive work.

Spyder IDE
Configuring

- For settings, see *Tools* ⊳ *Preferences*:



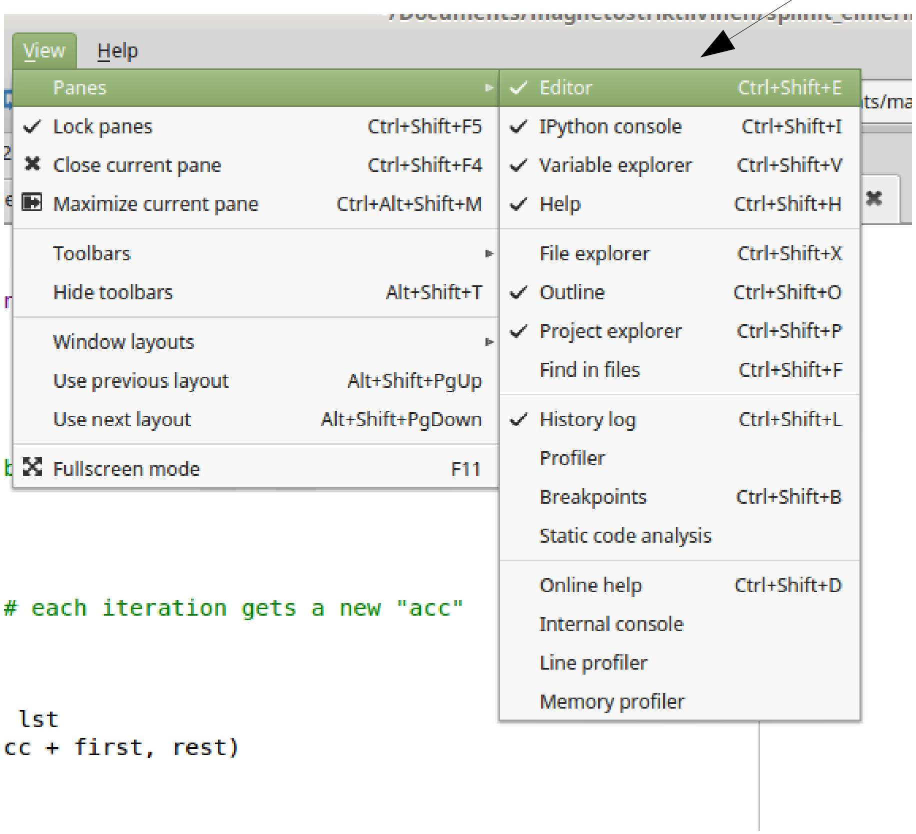
- One particularly useful option is whether figures open inline or in their own windows: *Tools* ⊳ *Preferences* ⊳ *IPython console* ⊳ *Graphics* ⊳ *Backend*

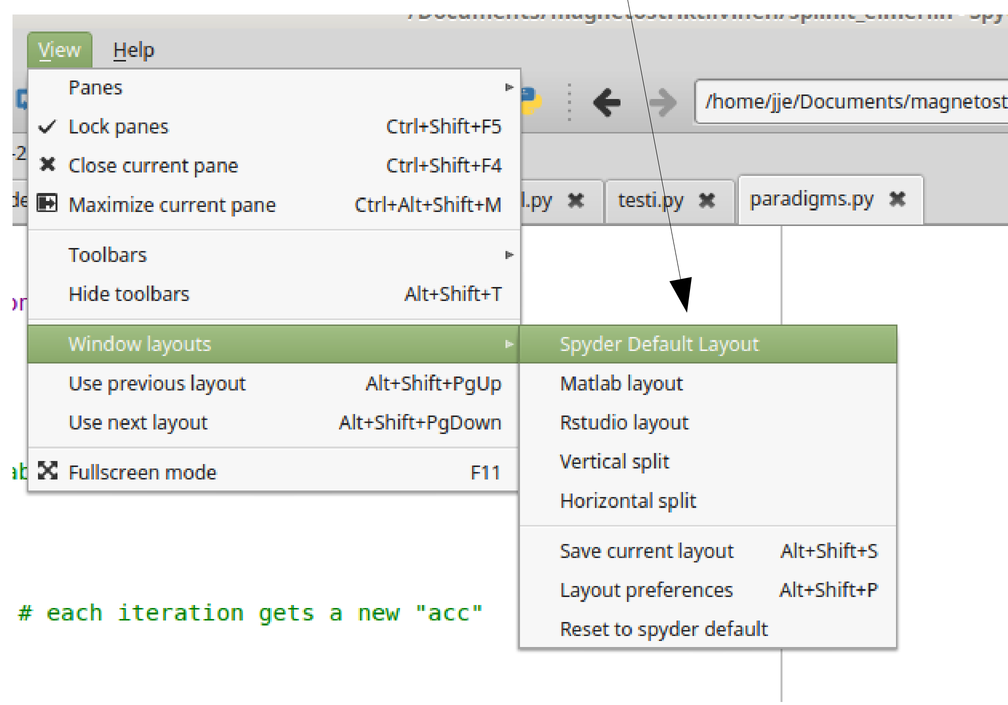
- Color setup for syntax highlighting: *Tools* ⊳ *Preferences* ⊳ *Syntax coloring* ⊳ *Scheme*

Spyder IDE

Customizing the layout

- If you want to change the layout, see *View ⊳ Panes* and *View ⊳ Window layouts*





To be continued...