

Introduction to Python

Part 1

v0.9

Research Computing Services
Information Services & Technology



About You

- Working with Python already?
- Have you used any other programming languages?
- Why do you want to learn Python?

RCS Team and Expertise

- Our Team
 - Scientific Programmers
 - Systems Administrators
 - Graphics/Visualization Specialists
 - Account/Project Managers
 - Special Initiatives (Grants)
- Maintains and administers the Shared Computing Cluster
 - Located in Holyoke, MA
 - ~23,000 CPUs running Linux
- Consulting Focus:
 - Bioinformatics
 - Data Analysis / Statistics
 - Molecular modeling
 - Geographic Information Systems
 - Scientific / Engineering Simulation
 - Visualization
- CONTACT US: help@scc.bu.edu

Python on the SCC

- There is documentation for using Python on the SCC on the [RCS website](#).
- Use the module system to find Python:

```
[bgregor@scc1 bg]$ module avail python3

----- /share/module.7/programming -----
python3-intel/2021.1.1    python3/3.7.3    python3/3.8.3
python3/3.6.5            python3/3.7.5    python3/3.8.6
python3/3.6.9            python3/3.7.7    python3/3.8.10.clean
python3/3.6.10           python3/3.7.9    python3/3.8.10      (D)
python3/3.6.12           python3/3.7.10   python3/3.9.4

Where:
D:  Default Module

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules matching
any of the "keys".
```

Python on the SCC

- Python can be used in qsub jobs by loading a Python module:

```
#!/bin/bash -l

#$ -P myproj
#$ -m ea
#$ -N PythonJob

module load python3/3.10.5

python myscript.py arg1 arg2
```

Getting Python for Yourself: Anaconda

- The most popular setup for personal computers
- <https://www.anaconda.com/download/>
- Anaconda is a packaged set of programs including the Python language, a huge number of libraries, and several tools.
 - These include the **Spyder** development environment and **Jupyter** notebooks.
- Anaconda can be used on the SCC, with [some setup required](#).

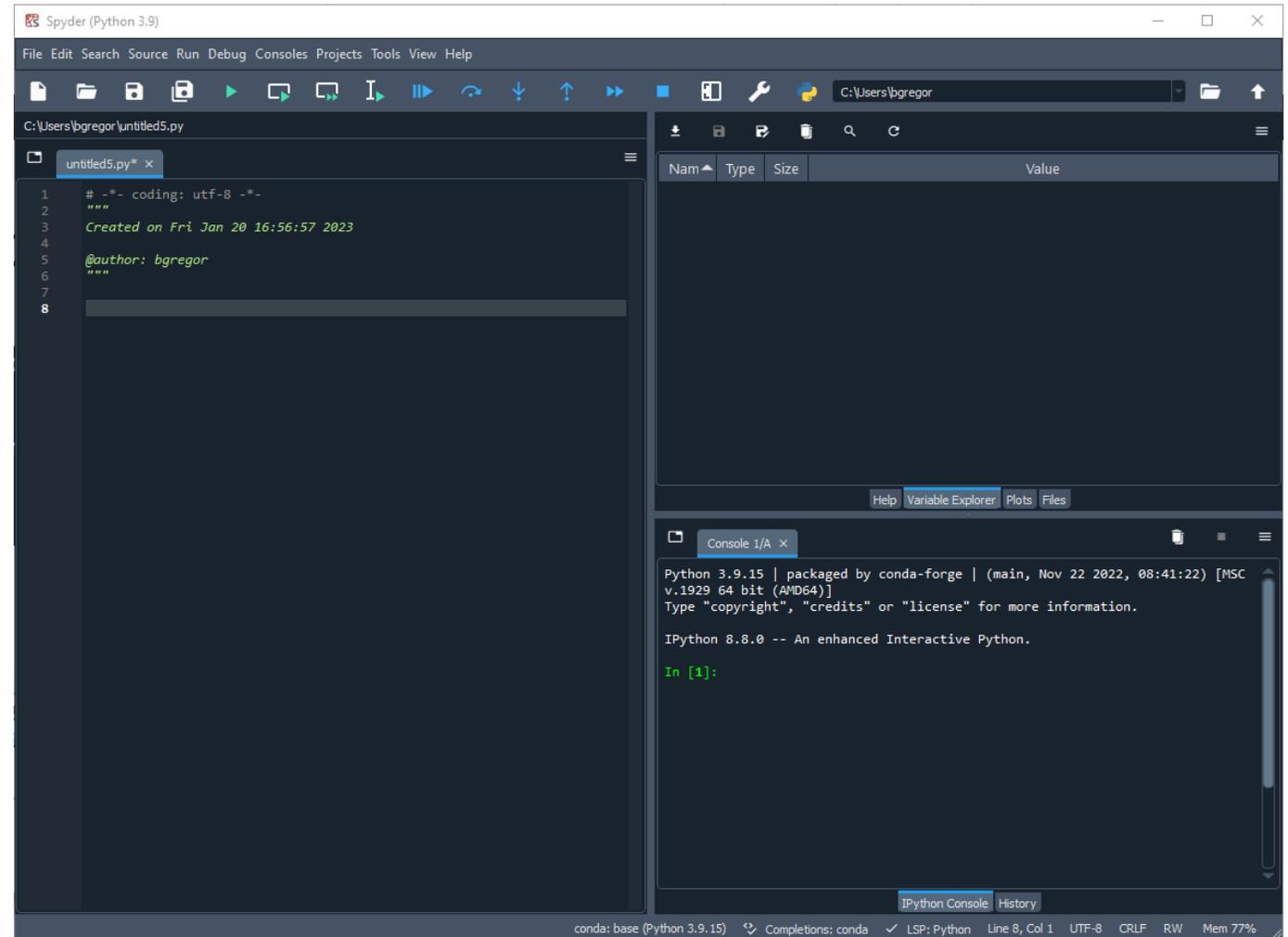
Spyder – a Python development environment

■ Pros:

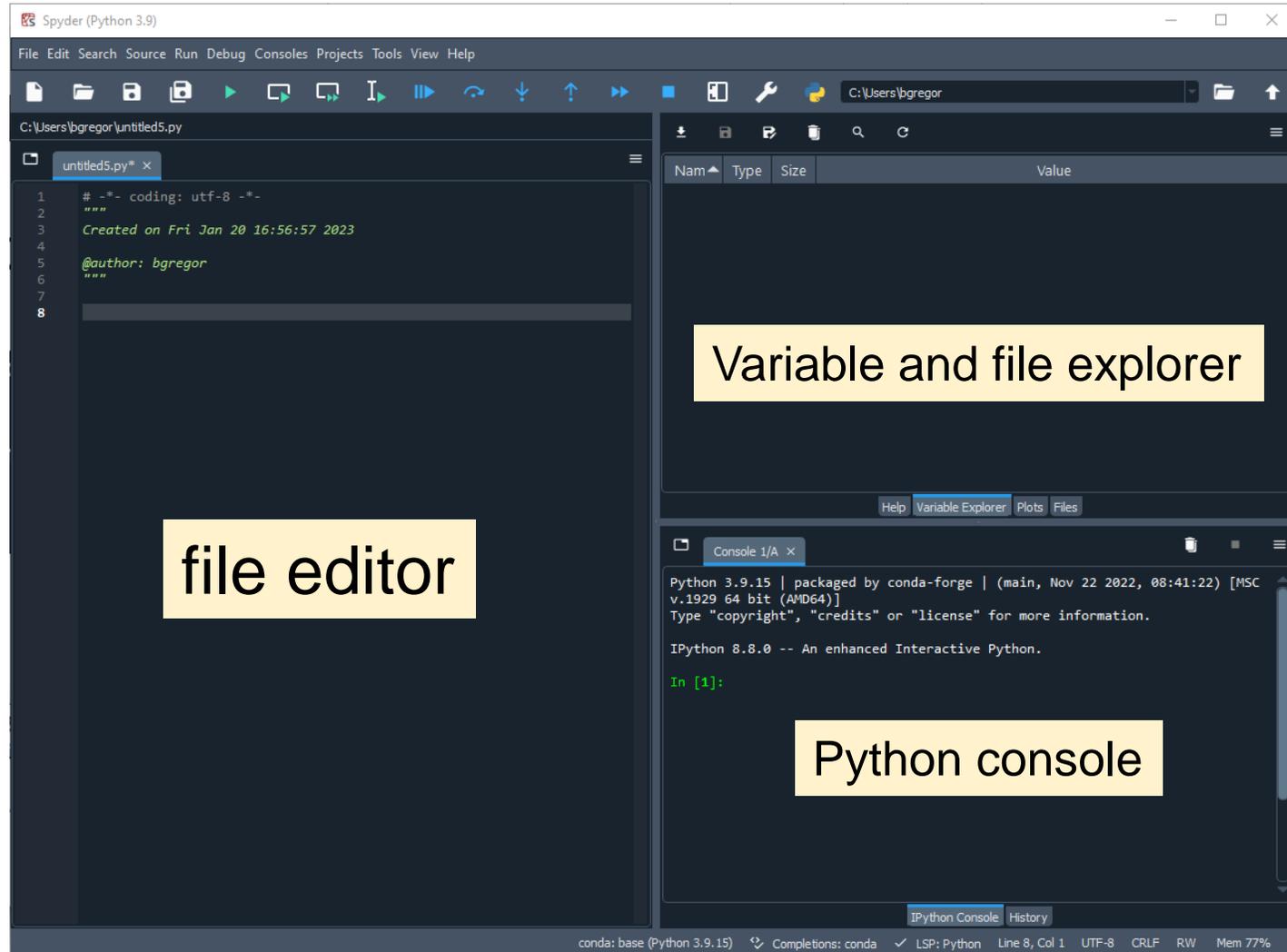
- Faster development
- Easier debugging!
- Helps organize code

■ Cons

- Learning curve
- Can add complexity to smaller problems



The Spyder IDE



Tutorial Outline – Part 1

- What is Python?
- Operators
- Variables
- Functions
- Lists

Some History

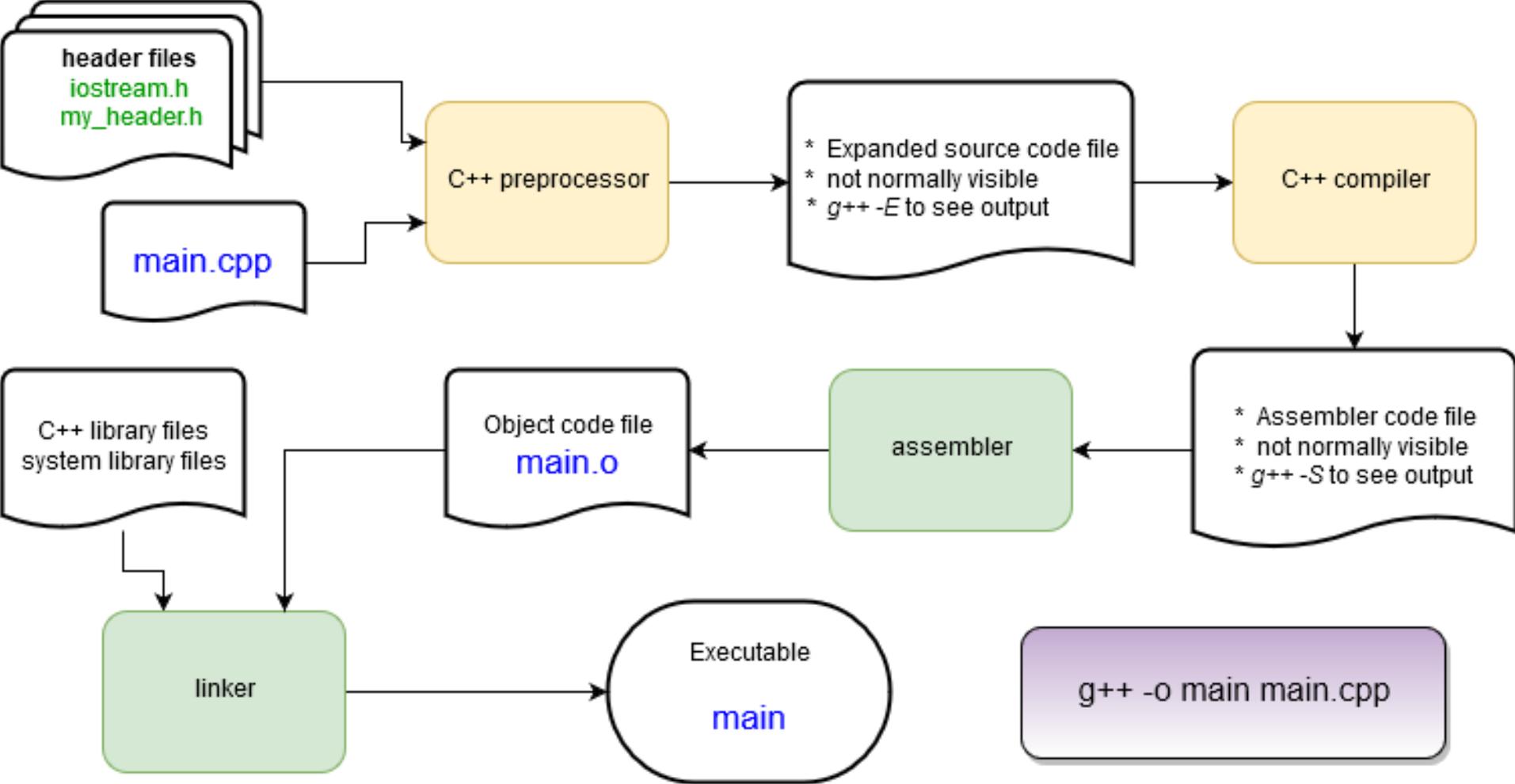
- “Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas...I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).”

–Python creator Guido Van Rossum, from the foreword to *Programming Python* (1st ed.)

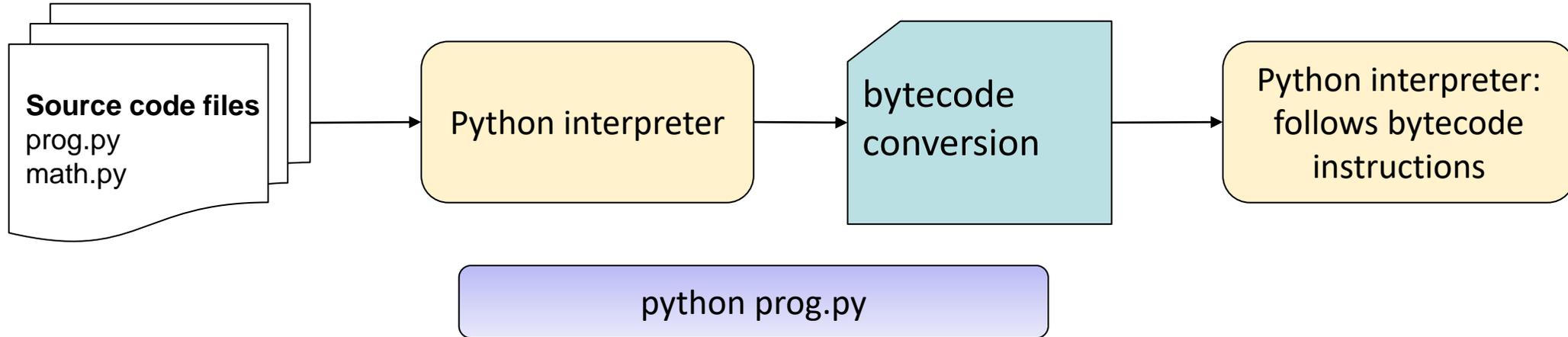
- Goals:
 - An easy and intuitive language just as powerful as major competitors
 - Open source, so anyone can contribute to its development
 - Code that is as understandable as plain English
 - Suitability for everyday tasks, allowing for short development times



Compiled Languages (ex. C++ or Fortran)



Interpreted Languages (ex. Python or R)



- A lot less work is done to get a program to start running compared with compiled languages!
- Python programs start running immediately – no waiting for the compiler to finish.
- Bytecodes are an internal representation of the text program that can be efficiently run by the Python interpreter.
- The interpreter itself is written in C and is a compiled program.

The Python Prompt

- The standard Python prompt looks like this:

```
[bgregor@scc2 bg]$ python
Python 3.6.2 (default, Aug 30 2017, 15:46:55)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-3)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

- The IPython prompt in Spyder looks like this:

```
Python 3.6.3 |Anaconda, Inc.| (default, Oct 15 2017, 03:27:45) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.1.0 -- An enhanced Interactive Python.

In [1]:
```

- IPython adds some handy behavior around the standard Python prompt.

Operators

- Python supports a wide variety of operators which act like functions, i.e. they do something and return a value:

- Arithmetic: + - * / // % **
- Logical: and or not
- Comparison: > < >= <= != ==
- Assignment: =
- Bitwise: & | ~ ^ >> <<
- Identity: is is not
- Membership: in not in

Try Python as a calculator

```
Python 3.9.15 | packaged by conda-forge | (main, Nov 22 2022, 08:41:22) [MSC
v.1929 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 8.8.0 -- An enhanced Interactive Python.

In [1]: 1 + 3
Out[1]: 4

In [2]: 4 * 2
Out[2]: 8

In [3]:
```

- Go to the Python prompt.
- Try out some arithmetic operators:

+ - * / // % ** == () and

- Can you identify what they all do?

Operators

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division ($25 / 4 = 6.25$)
//	Integer Division ($25 // 4 = 6$)
%	Remainder (aka <i>modulus</i>)
**	Exponentiation
==	Equals
and or not	Boolean operations
> < <= >=	Comparison

More Operators

- Try some comparisons and Boolean operators. *True* and *False* are the keywords indicating those values:

```
In [3]: 4 > 5
Out[3]: False

In [4]: 6 > 3 and 3 > 0
Out[4]: True

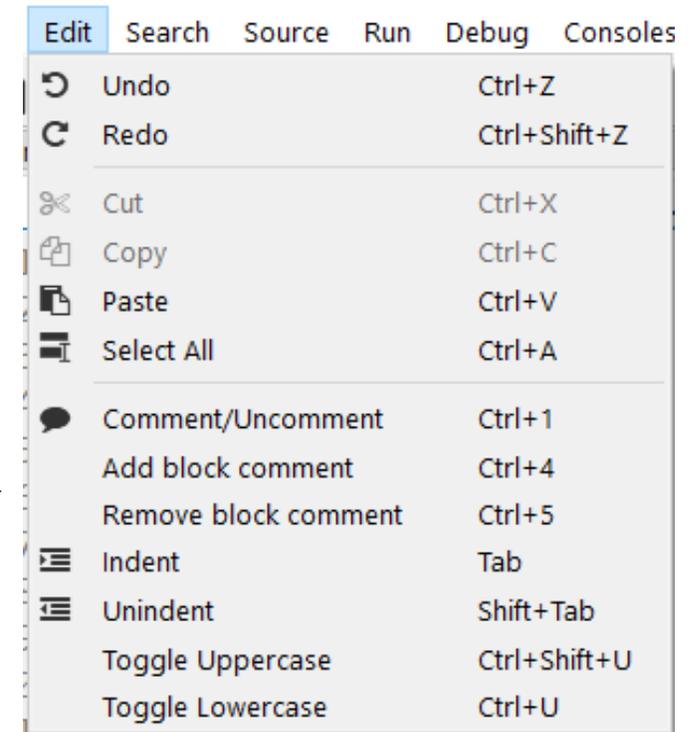
In [5]: not False
Out[5]: True

In [6]: True and (False or not False)
Out[6]: True
```

Comments

- # is the Python comment character. On any line everything after the # character is ignored by Python.
- There is no multi-line comment character as in C or C++.
- An editor like Spyder makes it very easy to comment blocks of code or vice-versa. Check the *Edit* menu

```
a=1
b=2
# this is a comment
c=3 # this is also a comment
# this is a
# multiline comment
```



Spyder Cells

- This is a Spyder-specific tool for helping you to run snippets of code in the file editor.
- Every time the characters `#%%` are seen Spyder treats that section as a “cell”.
- Right-click to run a single cell.
 - Or use the keyboard shortcuts

One cell

```
#%%  
# Import some libraries  
import math  
import sys  
  
#%%  
  
# Read a text file.  
with open('data.txt') as f:  
    all_data = f.readlines()  
  
#%%  
# Capitalize all the text that was read  
caps = [tmp.upper() for tmp in all_data]
```

Run cell	Ctrl+Return
Run cell and advance	Shift+Return
Re-run last cell	Alt+Return
Run selection or current line	F9
Run to current line	Shift+F9
Run from current line	Ctrl+F9
Go to definition	Ctrl+G
Inspect current object	Ctrl+I
Undo	Ctrl+Z
Redo	Ctrl+Shift+Z
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Select All	Ctrl+A
Zoom in	Ctrl++

Note: A “right-click” on a Mac is “click while holding down the Control key”

Variables

- Variables are assigned values using the = operator
- In the Python **console**, typing the name of a variable prints its value
 - **Not true in a script!**
 - [Visualize Assignment](#)
- Variables can be reassigned at any time
- Variable type is not specified
- Types can be changed with a reassignment

```
In [1]: a=1
```

```
In [2]: b=2
```

```
In [3]: a  
Out[3]: 1
```

```
In [4]: b  
Out[4]: 2
```

```
In [5]: a=b
```

```
In [6]: a  
Out[6]: 2
```

```
In [7]: b=-0.15
```

Variables cont'd

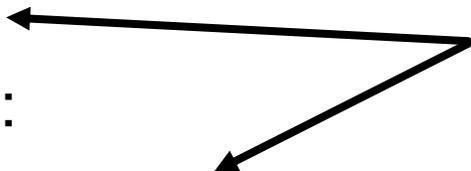
- Variables refer to a value stored in memory and are created when first assigned
- Variable names:
 - Must begin with a letter (a - z, A - Z) or underscore _
 - Other characters can be letters, numbers or _
 - Are case sensitive: capitalization counts!
 - Can be any reasonable length
- Assignment can be done *en masse*:

```
x = y = z = 1
```

- Multiple assignments can be done on one line:

```
x, y, z = 1, 2.39, 'cat'
```

Try these out!



Variable Data Types

- Python determines data types for variables based on the context
- The type is identified when the program **runs**, using **dynamic typing**
 - Compare with compiled languages like C++ or Fortran, where types are identified by the programmer and by the compiler **before** the program is run.
- Run-time typing is very convenient and helps with rapid code development

Variable Data Types

Numbers	Integers and floating point (64-bit)
Complex numbers	<code>x = complex(3,1)</code> or <code>x = 3+1j</code>
Strings	<code>"cat"</code> or <code>'dog'</code>
Boolean	<code>True</code> or <code>False</code>
Lists, dictionaries, sets, and tuples	These hold collections of values
Specialty types	Files, network connections, etc.
Custom types	User- or library-defined types using Python classes

Variable modifying operators

- Some additional arithmetic operators that modify variable values:

Operator	Effect	Equivalent to...
$x += y$	Add the value of y to x	$x = x + y$
$x -= y$	Subtract the value of y from x	$x = x - y$
$x *= y$	Multiply the value of x by y	$x = x * y$
$x /= y$	Divide the value of x by y	$x = x / y$

- The $+=$ operator is by far the most used of these.

Strings

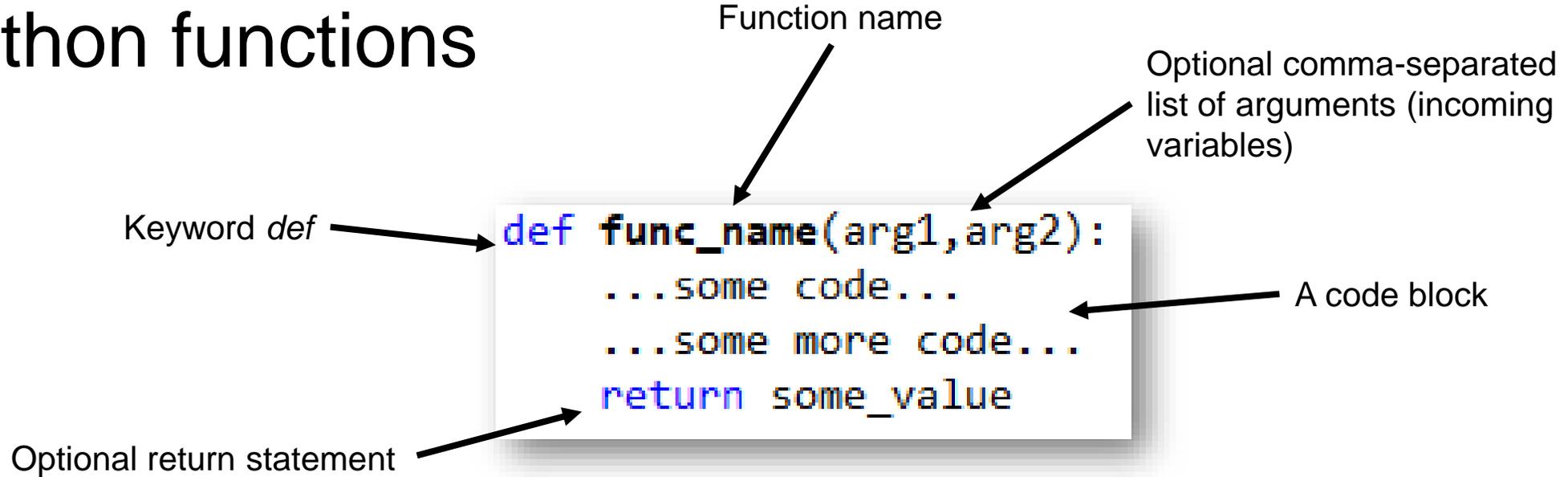
- Strings are a basic data type in Python.
- Indicated using pairs of single " or double "" quotes.
- Multiline strings use a triple set of quotes (single or double) to start and end them.

```
'cat'  
"dog"  
"What's that?"  
'They said "hello"'  
''' This is  
    a multiline  
    string '''
```

Functions

- Functions are used to create pieces of code that can be used in a program or in many programs.
- The use of functions is to logically separate a program into discrete computational steps.
- Programs that make heavy use of function definitions tend to be easier to:
 - develop
 - debug
 - maintain
 - understand

Python functions



- The return value can be any Python type
- If the return statement is omitted a special *None* value is still returned.
- The arguments are optional but the parentheses are required!
- **Functions must be defined** before they can be called.

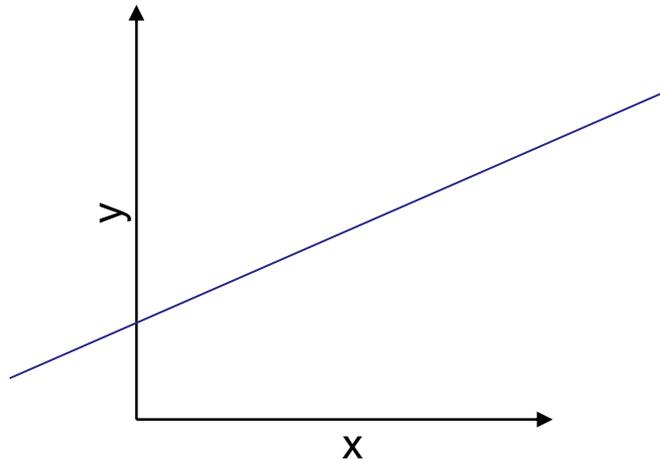
Sample Built-In Functions

- Let's try a few useful built-in functions...
- `print()`
- `dir()`
- `type()`
- `help()`

Visualize a Function Call

- Here's a simple function call to calculate the equation of a line.

$$y = A * x + B$$



Write a Function

- In Spyder's editor:

```
def mathcalc( ...your args here...):  
    # ... do something ...  
    return ...your calculated value...  
  
# Call your function and print the  
# result  
ans = mathcalc( ... args ...)  
print(ans)
```

- Define a function called **mathcalc** that takes 3 numbers as arguments and returns their sum divided by their product.
- Save the file and run it. Here's some sample output to check your result.

`mathcalc(1, 2, 3)` → returns 1.0

`mathcalc(4, -2.5, 3.0)` → returns -0.15

Which code sample is easier to read?

- C:

```
float avg(int a, int b, int c){  
float sum = a + b + c ;  
return sum / 3.0 ;}
```

or

```
float avg(int a, int b, int c)  
{  
    float sum = a + b + c ;  
    return sum / 3.0 ;  
}
```

- Matlab:

```
function a = avg(x,y,z)  
a = x + y + z ;  
a = a / 3.0 ;  
end
```

or

```
function a = avg(x,y,z)  
    a = x + y + z ;  
    a = a / 3.0 ;  
end
```

Which code sample is easier to read?

- Most languages use special characters (`{ }` pairs) or keywords (*end*, *endif*) to indicate sections of code that belong to:
 - Functions
 - Control statements like *if*
 - Loops like *for* or *while*
- Python instead uses the **indentation** that programmers use anyway for readability.

C

```
float avg(int a, int b, int c)
{
    float sum = a + b + c ;
    return sum / 3.0 ;
}
```

Matlab

```
function a = avg(x, y, z)
    a = x + y + z ;
    a = a / 3.0 ;
end
```

The Use of Indentation

- Python uses whitespace (spaces or tabs) to define *code blocks*.
- Code blocks are logical groupings of commands. They are **always** preceded by a colon :

```
def avg(x,y,z):  
    all_sum = x + y + z  
    return all_sum / 3.0  
  
def mean(x,y,z):  
    return (x + y + z) / 3.0
```

A code block

Another code block

- This pattern is consistently repeated throughout Python syntax.
- Spaces or tabs can be mixed in a file but **not** within a code block.

Function Return Values

- A function can return any Python value.
- Function call syntax:

```
A = some_func() # some_func returns a value
Another_func() # ignore return value or nothing returned
b,c = multiple_vals(x,y,z) # return multiple values
```

- Open *function_calls.py* for some examples

Function arguments

- Function arguments can be required or optional.
- Optional arguments are given a default value

```
def my_func(a, b, c=10, d=-1):  
    ...some code...
```

- To call a function with optional arguments:
- Optional arguments can be used in the order they're declared or out of order if their name is used.

```
my_func(x, y)           # a=x, b=y, c=10, d=-1  
my_func(x, y, z)        # a=x, b=y, c=z, d=-1  
my_func(x, y, d=w, c=z) # a=x, b=y, c=z, d=w
```

For Loops

- *For* loops are used to repeat commands a specified number of times.
- Python has a built-in function to produce a sequence of numbers, *range()*
 - `range(N)` → numbers 0 to (N-1)
 - `range(M, N)` → numbers M to (N-1)
 - `range(M, N, P)` → numbers M to (N-1) in steps of P
- Put that together with a *for* loop and run commands a specified number of times:

Indented code block, can be multiple lines long.

```
for i in range(10):  
    print(i)
```

range(10) → 0...9

i is first 0, then 1, then 2...

Project Euler Problem 6

- Write a function that solves this problem for an arbitrary amount N of natural numbers $(1,2,3,\dots,N)$
- In Spyder's editor write a function "euler6" that takes an argument N and returns this calculation:

Sum square difference

Problem 6

The sum of the squares of the first ten natural numbers is,

$$1^2 + 2^2 + \dots + 10^2 = 385$$

The square of the sum of the first ten natural numbers is,

$$(1 + 2 + \dots + 10)^2 = 55^2 = 3025$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$.

Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

```
def euler6(N):  
    # do your calculations here.  
    # hint: try a "for" loop...  
    # don't forget to return the result.  
    return ...your answer...  
  
# This should print 2640  
print(euler6(10))  
  
# This should print 25164150  
print(euler6(100))
```

Lists

- A Python list is a general purpose 1-dimensional container for variables.
 - i.e. it is a row, column, or vector of things
- Lots of things in Python act like lists or use list-style notation.
- Variables in a list can be of any type at any location, including other lists.
- Lists can change in size: elements can be added or removed

Making a list and checking it twice...

- Make a list with [] brackets.
- Append with the *append()* function
- Create a list with some initial elements
- Create a list with N repeated elements

Try these out yourself!
Add some `print()` calls to see the lists.

```
list_1 = []  
  
list_1.append(1)  
list_1.append('A string!')  
list_1.append([])  
  
list_2 = [4, 5, -23.0+4.1j, 'cat']  
  
list_3 = 10 * [42]
```

List functions

- Try `dir(list_1)`
- List have a number of built-in functions
- Let's try out a few...
- Also try the `len()` function to see how many things are in the list: `len(list_1)`

```
'append',  
'clear',  
'copy',  
'count',  
'extend',  
'index',  
'insert',  
'pop',  
'remove',  
'reverse',  
'sort']
```

List Indexing

- Elements in a list are accessed by an index number.
- Index #'s start at 0.
- List: `x = ['a', 'b', 'c', 'd', 'e']`
- First element: `x[0]` → 'a'
- Nth element: `x[2]` → 'c'
- Last element: `x[-1]` → 'e'
- Next-to-last: `x[-2]` → 'd'

List Slicing

```
x=['a', 'b', 'c', 'd', 'e']  
x[0:1] → ['a']  
x[0:2] → ['a', 'b']  
x[-3:] → ['c', 'd', 'e']  
# Third from the end to the end  
x[2:5:2] → ['c', 'e']
```

- Slice syntax: `x[start:end:step]`
 - The start value is inclusive, the end value is exclusive.
 - Start is optional and defaults to 0.
 - Step is optional and defaults to 1.
 - Leaving out the end value means “go to the end”
 - Slicing always returns a **new list copied from the existing list**

List assignments and deletions

- Lists can have their elements overwritten or deleted (with the *del*) command.
 - Note the *del* command does not use parentheses – it's sort of like a function call.

```
x=['a', 'b', 'c', 'd', 'e']
```

```
x[0] = -3.14 → x is now [-3.14, 'b', 'c', 'd', 'e']
```

```
del x[-1] → x is now [-3.14, 'b', 'c', 'd']
```

DIY Lists

- In the Spyder editor try the following things:
- Assign some lists to some variables. `a = [1,2,3]` `b = 3*['xyz']`
 - Try an empty list, repeated elements, initial set of elements
- Add two lists: `a + b` What happens?
- Try list indexing, deletion, functions from `dir(my_list)`
- Try assigning the result of a list slice to a new variable

More on Lists and Variables

- What happens when we pass a list to a function?
- Or we do an assignment with it?

```
def change_list(my_list, val):  
    if len(my_list) > 0:  
        first_val = my_list.pop(0)  
        my_list.extend([val, first_val])  
    return my_list  
  
x=[1,2]  
  
# call change_list, overwrite x  
x = change_list(x,10)  
  
# Do we need the return value?  
change_list(x, 20)  
  
# What about an assignment...  
y = x  
change_list(y,-1.5)  
print(x)
```

Copying Lists

- How to copy (2 ways...there are more!):
 - `y = x[:]` or `y=list(x)`
- Many data types in Python have this same behavior