Python 3.4 Quick Reference Guide V2.0

(for more info: https://docs.python.org/3/reference/)

Continuum Analytics Spyder editor default layout

- Left window is a tabbed edit window—create a file, save it, and click the green triangle to run it
- Right bottom window is a console—enter commands for immediate execution, and see output of programs

Line length: max 79 chars (up to vertical line)

- **Long line:** have bracket open at line end
- **Error-prone alternative:** put `\` at end of line
- **Comment:** any text after unquoted `#` on a line

Data Types, Literals and Conversions

- **Integers**: optional sign with digits (no limit)
  `0, 314, -71322956627718, +6122233`
  `int(-5.7) → -5 # remove decimal part`
- **Floats**: decimal fraction, exponent (~16 digits)
  `0., 3.141592653589793, -7.132E8, 9.9e-20`
  `float(3) → 3.0 # convert to float`
- **Complex**: `z.real` and `z.imag` are always floats
  `0.0j, 3-7.132E8j, z = 2.1+0.9j`
  `complex(x,y) → x + yj # from 2 floats`
- **String**: single or double quotes allowed
  `'X', "I'd", '"No," he said.'`
  `str(1 / 93) → '0.010752688172043012' # converts`
  `"\N{MICRO SIGN}" → 'µ' # Unicode by name`
- **Multi-line string**: triple quotes (single or double)
  `"""The literal starts here and goes on 'line after line' until three more quotes."""`
- **Boolean**: two aliases for 1 and 0 respectively:
  `bool(x) → True` or `False`
  Any zero or empty value can be used as `False` in a boolean expression; other values mean `True`
- `type("age") → str ; type(3.14) → float`

Math operators:

- except for /, give `int` type result if *x* and *y* are both `int`
- give `float` type result if *x* or *y* is `float`
- Power (x^y), Times ($x \times y$): `x**y, x * y`
- Divide ($x \div y$): true divide `x / y` or floor divide `x // y`
  `//` result *value* is integer, but *type* may not be `int`;
  `/` result is always `float`, in both type and value
- Remainder (*x* mod *y*): `x % y`
- Add (*x*+*y*), Subtract (*x*–*y*): `x + y, x - y`

Operators with bool result

- Compare: `<, <=, !=, ==, >=, >`
- `x > y` → either `True` or `False` (1 or 0)
- `3 <= x < 5` means `3 <= x and x < 5`
- `x is y` means `x`, `y` refer to the same object (T/F)
- `x in y` means `x` is found inside `y` (T/F)

Identifiers

- **Variables** (mixed case), **Constants** (all uppercase)
  `sumOfSquares = 0.0 # [-] value can be changed`
  `SECS_PER_MIN = 60  # [s/min] should be fixed`

Evaluation Order from High Priority to Low

- `**`: `-2**2**3` is like `-(2**(2**3))`
- `+,-`: `-++-+3` is like `-(+(+(-(+3))))`
- `*,/,//,%`: `8 / 3 // 2 * 3 % 2` is like
  `(((8 / 3) // 2) * 3) % 2`
- `+,-`: `8 / 3 * 4 + -2**4 % 5` is like
  `((8 / 3) * 4) + ((-(2**4)) % 5)`
- `<,<=,!=,==,>=,>,is,in`:
  `5 + 2 < 3 * 8` is like `(5 + 2) < (3 * 8)`

Local and global variables

- a variable defined in the Python console is *global*
- you can access it and *use* its value in a function
- a variable *assigned a value* in a function definition is *local*; it exists only during function execution, unless you *declare* a variable global inside the function:
  `global CHARS; CHARS = list("abc")`

Assignment

- `x = y` makes identifier `x` refer to the same object that `y` refers to
- `x,y = a,b` is like `x = a; y = b # done simultaneously`
- `x,y = y,x` swaps values of `x` and `y`
- `x += a` is like `x = x + a`
- `x -= a` is like `x = x - a`
- similarly for `*=, /=, //=, %=, **=`

Output with old style formatting

- The call `print(3,5,(1,2))` displays blanks between
  `3 5 (1, 2)`
- `"%d and %f: %s" % (3,5.6,"pi")`
  `→ "3 and 5.600000: pi"`
- Conversion specifiers:

`%s`	string version, same as `str(x)`
`%r`	best representation, same as `repr(x)`
`%c`	shows number as a character, same as `chr(x)`
`%d`	an integer
`%04d`	an integer left padded with zeros, width 4
`%f`	decimal notation with 6 decimals
`%e, %E`	scientific notation with **e** or **E**
`%g, %G`	compact number notation with **e** or **E**

- Meaning of format qualifiers for any format type `<f>`

`%8f`	format right-adjusted, width 8
`%-9d`	format left-adjusted, width 9
`%20.7f`	7 decimals (7 sig. digits for **g** format), width 20
`%%`	a literal % sign

Input from user

- `var = input("Enter value: ")`
- `intVal = int(var) # integer cast, or`
- `anyVal = eval(var) # could be dangerous, or`
- `assert '@' in var, "%s not e-mail address" % var`

Operating system functions:

```
import os, sys # operating system, system
print(os.listdir('.')) # file list of current folder
os.chdir("A1") # change folder to A1
```

Built-in functions: `dir(__builtins__)`

- `abs(x) → |x| # works on complex too`
- `chr(35) → '#'; chr(169) → '©'`
- `dir(x)` → attributes of `x`
- `help(x)` → help on using `x`
- `len(x)` → length of `x`
- `max(2.1, 4, 3) → 4     # largest argument`
- `min(2.1, 4, 3) → 2.1   # smallest argument`
- `ord('#') → 35  # ASCII order`
- `range(a,b,c)` *see lists*
- `round(3.6) → 4 # nearest int`
- `round(3.276,2) → 3.28 # 2 decimals`
- `sum( (1, 5.5, -8) ) → -1.5 # add items`
- `zip(listx, listy)` → list of (*x*,*y*) tuples

Math functions: `dir(math)` for list

```
import math as m  # for float, or
import cmath as m # for complex
```

- `m.acos(x)` → inverse cosine [radians]
- `m.asin(x), m.atan(x) # similar`
- `m.ceil(x)` → least integer ≥ `x` [math only]
- `m.cos(x)` → cosine of `x` given in radians
- `m.e` → 2.718281828459045
- `m.exp(x)` → e^x
- `m.factorial(x)` → `x!` [math only]
- `m.floor(x)` → biggest int ≤ `x` [math only]
- `m.log(x)` → natural log of `x`
- `m.log10(x)` → log base 10 of `x`
- `m.pi → 3.141592653589793`
- `m.sin(x), m.tan(x) # see m.cos`
- `m.sqrt(x)` → square root of `x`

`import random # for pseudorandom numbers`

- `random.random()` → uniform in [0,1)
- `random.seed(n) # resets random number stream`
- `random.randrange(a,b)` → uniform int in [a,b)

`while` control structure

- `while` statement followed by indented lines
- `while` *condition* :
  —loops while *condition* is `True` (i.e., not 0)
  —indented lines are repeated each iteration
  —to terminate, make *condition* 0/`False`/empty
-
```
leftToDo, total = 100, 0.0
while leftToDo :  # i.e., while leftToDo > 0:
    total    += 1.0 / count
    leftToDo -= 1
```

Sets

- A set is a collection of unique values, using `{ }`
- `a = {1,2}; b = {1,5}`
- `a.union(b) → {1,2,5} # also a | b`
- `a.intersection(b) → {1} # also a & b`
- a – b → {2} ; b – a → {5} `# in one, not the other`

Lists and tuples

- Both are sequences of arbitrary objects with index positions starting at 0, going up by 1
- A tuple has optional round brackets
- `xt = 1,    # a single member tuple`
- `xt = (1, 8.9, "TV")`
- A list has square brackets (required)
- `xl = [1, 8.9, "TV"]`
- `list("too") → ['t','o','o']`
- `tuple(["age",5]) → ('age',5)`

What you can do with a tuple

- You cannot change length or contents (***immutable***)
- `len(xt) → 3`
- `8.9 in xt → True`
- `xt.count(8.9) → 1 # how many 8.9 entries`
- `xt.index("TV") → 2 # first TV in position 2`
- `xt[2] → 'TV'    # in position 2`
- `xt[-1] → 'TV'   # in last position`
- `xt[0:2] → (1, 8.9)    # extract slice`
- `xt[1:] → (8.9, 'TV')  # open-ended slice`
- `xt[0: :2] → (1, 'TV') # slice by twos`

What you can do with a list

- almost anything you can do with a tuple, plus...
- `xl.append(5.7)   # adds 5.7 to end`
- `xl.insert(3,-3) # put -3 before xl[3]`
- `xl.pop(2) → 'TV' # remove 3rd entry`

  → `xl` is now `[1, 8.9, -3, 5.7]`
- `xl.reverse() # reverses order`
- `xl.sort()    # in increasing order`

  → `xl` is now `[-3, 1, 5.7, 8.9]`
- `xl[1] += 2.2   # updates entry value`
- `xl[:2] = [2,3] # assign to slice`
- `xl[:] → [2, 3, 5.7, 8.9] # a copy`
- `range` is a generator: it produces a sequence of values

  ```
  list(range(5)) → [0, 1, 2, 3, 4]
  list(range(2,5)) → [2, 3, 4]
  list(range(2,15,3)) → [2, 5, 8, 11, 14]
  ```

Third element of a slice is a step size:

- `range(50)[::9] → range(0,50,9)`

`for` control structure

- `for` statement followed by indented lines
- `for item in listOrTupleOrStringOrSetOrDict :`

  —`item` takes on the value of each entry in turn

  —indented lines are repeated for each `item`

-
  ```
  total = 0.0
  for number in range(1,101):
      total += 1.0 / number
  ```
- Parallel lists `list1` and `list2`, or position and value:

  ```
  for e1, e2 in zip(list1, list2):
      print(e1, e2)
  for pos, value in enumerate(list1):
      print("Entry at %d is %g" % (pos,value))
  ```

List comprehension (embedded `for`)

- To generate one list from items in another
- `list2 = [f(n) for n in list1]`
- `squares = [n * n for n in range(90)]`

`if - else` blocks

- `if` *condition* :   `# if statement`

  indented lines
- `elif` *condition* : `# elif optional, repeatable`

  indented lines
- `else` :   `# else optional`

  indented lines
- Python executes just the first section that has a `True` condition, or else the `else` statement if present

Dictionaries

- A dictionary is a collection of (key, value) pairs
- `wordCnts = {}   # creates a new dict`
- `for word in text.split() :`
- `    if word in wordCnts : # efficient check`
- `        wordCnts[word] += 1`
- `    else :`
- `        wordCnts[word] = 1`
- `wordCnts.keys()   # generator: keys in dictionary`
- `wordCnts.values() # corresponding value generator`
- `wordCnts.items()  # generator: (key, value) pairs`

Defining functions

- `def` statement followed by indented lines
- `def myFunc(parm1, parm2, …) :`

  —creates a function `myFunc` with parameters

  —*`parmj`* is given the value used in calling `myFunc`

  —indented lines are executed on call

  —`return y` returns the value `y` as the results of the call

-
  ```
  def vectorLength(x,y):
      return m.sqrt(x * x + y * y)
  ```
- `vectorLength(3,4) → 5.0`

-
  ```
  def first3Powers(n): # multiple returns
      return n, n * n, n**3 # returns a 3-tuple
  ```
- `x1, x2, x3 = first3Powers(6.2)`

Strings: convert to string, and manipulate them

- `str(x)` → default string representation of `x`
- `"banana".count('an') → 2 # number of occurrences`
- `"banana".find('an') → 1 # first location`
- `"banana".find('an', 2) → 3 # location after 2`
- `"banana".find('bn') → -1 # not found`
- `'%'.join(['a', 'b', 'c']) → 'a%b%c'`
- `"abcb".replace('b',"xx") → 'axxcxx'`
- `"a bc d ".split() → ['a', 'bc', 'd']`
- `"a bc d ".split('b') → ['a ', 'c d']`
- `"  ab c ".strip() → 'ab c'`
- `"'ab'c'".strip("'") → "ab'c"`
- `"200 Hike!".upper() → '200 HIKE!'`
- `"200 HIKE!".lower() → '200 hike!'`

File reading from file in current directory

- Open file to read (`'r'`) with any newline char

  `flink = open(file,"r",newline=None)`
- `text = flink.read()  # read entire file`
- or read file line by line:

  ```
  text = ""             # initialize input
  for line in flink :   # read line
      text += line      # save line
  ```
- `flink.close()        # close file when done`

File reading from url

-
  ```
  import urllib.request
  url = "http://sample.org/file.txt"
  flink = urllib.request.urlopen(url)
  # continue as when reading file
  ```

Numpy arrays

`import numpy as np`

A collection of ***same-type*** objects, with functions:

- `np.arange(a,b,c)  # range-like, yields array`
- `np.array(x) # make copy, or convert x`
- `np.linspace(0.1,0.8,n) # n floats`
- `np.repeat(x,n) # repeat each entry n times`
- `np.resize(x,shape) # fit into new shape`
- `np.zeros(n), np.ones(n)# n floats`
- `np.random.random(n) # n values in [0,1)`
- `np.random.randint(a,b,n) # int in [a,b)`
- `np.random.seed(s) # reset seed to s`
- `np.random.shuffle(x) # shuffle x`

Math operations with arrays:

- `+ - * / // % **` operations are done *item by item*
- `np.int_(x) # casts to int`
- *vectorized* math functions (e.g., `np.sqrt`, `np.sum`) handle real or complex array inputs
- `n1 = np.arange(5); n2 = n1[1::2]`
- `n2[:] = 4; n1 → [0, 4, 2, 4, 4]`

Plotting

- `import matplotlib.pyplot as plt`
- `fig = plt.figure(n) # which figure to alter`
- `fig.add_subplot(326,aspect="equal")`

  `# 3x2 rows, columns, 6’th subplot`
- `plt.plot(x,y,fmt,label="…") # plots y vs x`

  like `scatter`, `semilogx`, `semilogy`, `loglog`
- `fmt` (optional) is a string with colour and type:

  —`r` (red), `g` (green), `b` (blue), `k` (black)

  —`o` (circle), `-` (solid), `--` (dashed), `:` (dotted)

  —`<`,`v`,`>` (triangles), `s` (square), `*` (star)
- `plt.xlabel`, `plt.ylabel`, `plt.title`,

  `plt.legend(loc = "best") # add legend to plot`
- `plt.savefig # makes image file of plot`
- `plt.hist(xx, 20) # plots histogram in 20 bins`
- `plt.xlim(min, max) # sets limits on x-axis`
- `plt.ylim(min, max) # sets limits on y-axis`
- `plt.show() # makes plot appear on the screen`