Minimalist Python Notes **Wendy Olsen Feb. 2026, v1.0**

This informal Python introducer helps you see a few details of the wording, layout and results from typical early Python runs. You type inputs, run the input in chunks, and look at your output.

A line continuation marker is an ellipsis at the start of the next line. … for example, in the 2nd line, … `This continues.`

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this if statement:

```
>>>
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
```

Be careful not to fall off! I did it with 2 line breaks of course:

```
print("Rubbish, it's not flat")
...
```

Source: The online Python book, 2. Using the Python Interpreter — Python 3.10.18 documentation I'd like to thank the authors of this online resource. See URL https://docs.python.org/3.10/tutorial/interpreter.html , accessed Feb. 2026

The Python language is named after the BBC show "Monty Python's Flying Circus" and has nothing to do with reptiles. Making references to Monty Python skits in documentation is not only allowed, it is encouraged!

From Tutorial for 3.8, https://docs.python.org/3.8/tutorial/appetite.html

CTRL-D to quit. (Unix)

CTRL-Z to quit when in Windows.

Perhaps the quickest check to see whether command line editing is supported is typing `Control-P` to the first Python prompt you get. If it beeps, you have command line editing ; if nothing appears to happen, or if `^P` is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

Or use this from shell:

Python -c command [arg]

Wise to put the commnd into " " quotes. That's due to spaces in it.

Or, run a py module as a script, using

Python -m module [arg]

When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing `-i` before the script.

It uses UTF-8 encoding.

BEWARE NUMPY package is needed before you run these:

```
import numpy as np

import pandas as pd

pd.options.display.max_columns = 20

pd.options.display.max_rows = 20

pd.options.display.max_colwidth = 80

np.set_printoptions(precision=4, suppress=True)
```

the last printed expression is assigned to the variable _

The ** is exponentiation

```
iwant = "Doesn\'t"

>>> iwant

"Doesn't"

>>> print(iwant)

Doesn't

>>>
```

Use \n to indicate a fresh line start.

To input a raw string without \ markers, use r before the first quote, eg command r"C:\data\input.csv" will look for that file. But otherwise: "C:\\data\\input.csv"

+ is concatenate for strings, '2' + '49' gives 249

Auto concatenate such as 'hhid' 'pid' gives hhidpid

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
```

If you want to concatenate variables or a variable and a literal, use `+`:

```
>>>
```

```
>>> prefix + 'thon'
'Python'
```

A list needs commas to separate items.

The given end-point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even a negative increment; this third argument is called the 'step'):

```
range(5, 10)
   5, 6, 7, 8, 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Argument: an input to a function. Here, 'range' is the funcation and start, stop, step are the three arguments.

Parameter: a word that refers to inputs like arguments, scalars (ie numbers) and other variables. A typical parameter would be x. Another is y and a third, i.

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

The for loop uses a parameter i. Then, i is also the first argument in the print command. It's also the subscript on the vector a. The vector a is specified as a Pandas List using [] at the assignment stage. = is the assignment operator.

```
list(range(4))
[0, 1, 2, 3]
```

The range command in base Python is actually a method. It can be properly written range() because it does require one parameter. So range(5) gives [0, 1, 2, 3, 4] in python. Can you see the pattern? Its numbering system starts at 0 in lists.

The keyword `def` introduces a *new function definition*. Def must be followed by the function name and the parenthesized list of formal parameters. Next there is a colon. (:) The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal. This string literal is the function's documentation string, or *docstring*. (More about docstrings can

be found in the section Documentation Strings.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

In a large organization, creating User-Defined Functions may be a valuable activity. There are alternatives, but learning to make a UDF is a great step on the ladder of python learning. A later step is known as making a class. For example a class of entities would enable several methods, suited to that class, and you would write code to define the 1 class and perhaps 5 relevant methods for that class.

The *execution* of a function can use default parameters for the function, eg ask_ok(). Usually we might want to declare the parameter value(s). For example, ask_ok(`'Do you really want to quit?'`)

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function could be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

To set a default when defining a UDF, we use the = again. We set the default at the right of = and the name of the argument to the left of =. So we may write:

```
def ask_ok(prompt='Do you really want to quit?', retries=4,
reminder='Please try again!'):
```

etc. Then the user wouldn't have to type all that text in, but could just start the function by writing ask_ok().

Functions can also be called using keyword arguments of the form `kwarg=value`. A kwarg is a keyword argument. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='voom',
type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      ----------    ----------     ----------
        |             |                  |
        |        Positional or keyword   |
        |                                - Keyword only
         -- Positional only
```

where `/` and `*` are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also referred to as named parameters.

To mark parameters as *keyword-only*, indicating the parameters must be passed by keyword argument, place an `*` in the arguments list just before the first *keyword-only* parameter.

FUNCTIONS CAN HAVE DEFAULT ARGUMENTS, or other kinds of arguments.

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>,
'eggs': <class 'str'>}
Arguments: spam eggs
```

```
'spam and eggs'
```

Here, ham is the default for the 1st string argument.

And --> str means the arguments must both be strings.

What do I mean: I mean that the colon ends the whole definition line, and before the colon, you can assign a return value. Watch:

"Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement"

Here the return value is to be taken overall as a string, it says. So it concatenates.

PEP8 style:

4 spaces at left if indent. 79 char at most at right.

Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4).`

The PEP8 is Python's style guide for programmers. The python language official style guide PEP8 is the eighth Python Enhancement Proposal. It is given here, and was first written by the creator of python Guido van Rossum.

R does not have one single agreed upon style guide, but a commonly used one is the Google Style Guide. This is linked to here.

Both of these guides are definitely worth a read at some point if you are trying to write better code.

See also Source: ONS, 2025. Unit Testing in Python – Unit Testing accessed Aug. 2025.

Reference:

The online Python book, 2. Using the Python Interpreter — Python 3.10.18 documentation I'd like to thank the authors of this online resource. See URL https://docs.python.org/3.10/tutorial/interpreter.html , accessed Feb. 2026