A Quick Intro to Python Programming

Scientific computing with a kinematics example

Manuel A. Diaz

InspiraSTEM Internship

July 21, 2026

Outline

This is a very quick intro to Python programming

- variables for numbers, lists, and arrays
- while loops and for loops
- functions
- if tests
- plotting
- files
- classes

Method: show program code through math examples

Outline

Do you have access to Python?

For this course we use uv to install Python and packages.

1. Install `uv`
2. At the project root (folder with `pyproject.toml`):

```
uv sync
cd python_intro
uv run python distance.py
```

`uv run` uses the project environment; no manual `venv` activation needed.

Mathematical example

Most examples will involve this formula:

$$s = v_0 t + \frac{1}{2} a t^2 \tag{1}$$

We may view s as a function of t: $s(t)$, and also include the parameters in the notation: $s(t; v_0, a)$.

A program for evaluating a formula

Task: Compute s for $t = 0.5$, $v_0 = 2$, and $a = 0.2$.
Python code (`python_intro/distance.py`):

```
t = 0.5
v0 = 2
a = 0.2
s = v0 * t + 0.5 * a * t**2
print(s)
print(f's={s:g}')
print(f's\t = \t {s:.3f}')
```

Execution

```
Terminal> uv run python distance.py
1.025
s=1.025
s = 1.025
```

Assignment statements assign a name to an object

```
t = 0.5 # real number makes float object
v0 = 2 # integer makes int object
a = 0.2 # float object
s = v0*t + 0.5*a*t**2 # float object
```

Rule:

- evaluate right-hand side; it results in an *object*
- left-hand side is a name for that object

Formatted output with text and numbers

- Task: write out text with a number (3 decimals): `s=1.025`
- Method: f-strings (modern Python)

```
print(f's={s:g}') # g: compact notation
print(f's={s:.2f}') # f: decimal notation, 2 decimals
```

Older alternatives (still valid):

```
print('s=%g' % s)
print('s={s:.2f}'.format(s=s))
```

Programming with a while loop

- Task: write out a table of t and $s(t)$ values (two columns), for $t \in [0, 2]$ in steps of 0.1
- Method: while loop

```
v0 = 2
a = 0.2
dt = 0.1 # Increment
t = 0 # Start value
while t <= 2:
    s = v0 * t + 0.5 * a * t**2
    print(t, s)
    t = t + dt
```

Output of the previous program

```
Terminal> uv run python while.py
0 0.0
0.1 0.201
0.2 0.404
0.3 0.609
0.4 0.816
0.5 1.025
...
1.8 3.924
1.9 4.161
```

Structure of a while loop

```
while condition:
    <indented statement>
    <indented statement>
    <indented statement>
```

Note:

- the colon in the first line
- all statements in the loop *must be indented* (no braces as in C, C++, Java, ...)
- `condition` is a boolean expression (e.g., `t <= 2`)

Closer look at the output

```
Terminal> uv run python while.py
0 0.0
0.1 0.201
...
1.8 3.924
1.9 4.161
```

The last line contains 1.9, but the while loop should run also when $t = 2$ since the test is `t <= 2`. Why is this test `False`?

Examine the program (Python Tutor idea)

Step through a smaller loop and examine variables (Python Tutor):

```
a = 0
da = 0.4
while a <= 1.2:
    print(a)
    a = a + da
```

Round-off errors!

Why is `a <= 1.2` false when `a` "is" 1.2?

```
a = 0
da = 0.4
while a <= 1.2:
    print(a)
    a = a + da
    print(f'da={da:.16E}\na={a:.16E}')
    print(f'a <= 1.2: {a <= 1.2}')
```

Key point

Small round-off error in `da` makes `a = 1.2000000000000002`

Never `a == b` for real numbers

Always use a tolerance!

```
a = 1.2
b = 0.4 + 0.4 + 0.4
boolean_condition1 = a == b # may be False

# This is the way to do it
tol = 1E-14
boolean_condition2 = abs(a - b) < tol # True
```

A list collects several objects in a sequence

A list of numbers:

```
L = [-1, 1, 8.0]
```

A list can contain any type of objects:

```
L = ['mydata.txt', 3.14, 10] # string, float, int
```

Basic list operations

```
>>> L = ['mydata.txt', 3.14, 10]
>>> print(L[0]) # first element (index 0)
mydata.txt
>>> print(L[1]) # second element (index 1)
3.14
>>> del L[0] # delete the first element
>>> print(L)
[3.14, 10]
>>> print(len(L)) # length of L
2
>>> L.append(-1) # add -1 at the end
>>> print(L)
[3.14, 10, -1]
```

Store our table in two lists

```
v0 = 2
a = 0.2
dt = 0.1 # Increment
t = 0
t_values = []
s_values = []
while t <= 2.1:
    s = v0 * t + 0.5 * a * t**2
    t_values.append(t)
    s_values.append(s)
    t = t + dt
print(s_values) # Just take a look at a created list
print(t_values)

# Print a nicely formatted table
i = 0
while i <= len(t_values) - 1:
    print(f'{t_values[i]:.18f} {s_values[i]:.4f}')
    i += 1 # Compact form for i = i + 1
```

For loops

A for loop visits elements in a list, one by one:

```
>>> L = [1, 4, 8, 9]
>>> for e in L:
... print(e)
...
1
4
8
9
```

For-loop demo (Python Tutor style)

```
list1 = [0, 0.1, 0.2]
list2 = []
for element in list1:
    p = element + 2
    list2.append(p)
print(list2)
```

Integer counter over list/array indices

```
somelist = ['file1.dat', 22, -1.5]

for i in range(len(somelist)):
    # access list element through index
    print(somelist[i])
```

Note:

- `range` returns a sequence of integers
- `range(a, b, s)` gives `a, a+s, ...` up to *but not including* b
- `range(b)` implies $a = 0$ and $s = 1$
- `range(len(somelist))` gives $0, 1, 2$

Replace our while loop by a for loop

```
v0 = 2
a = 0.2
dt = 0.1 # Increment
t_values = []
s_values = []
n = int(round(2 / dt)) + 1 # No of t values
for i in range(n):
    t = i * dt
    s = v0 * t + 0.5 * a * t**2
    t_values.append(t)
    s_values.append(s)
print(s_values) # Just take a look at a created list

# Make nicely formatted table
for t, s in zip(t_values, s_values):
    print(f'{t:.2f} {s:.4f}')

# Alternative implementation
for i in range(len(t_values)):
    print(f'{t_values[i]:.2f} {s_values[i]:.4f}')
```

Traversal of multiple lists with `zip`

```
for e1, e2, e3, ... in zip(list1, list2, list3, ...):
```

Alternative: loop over a common index

```
for i in range(len(list1)):
    e1 = list1[i]
    e2 = list2[i]
    e3 = list3[i]
```

Arrays are efficient lists of numbers

- Lists collect objects in a single variable
- Lists are flexible (can grow, can contain “anything”)
- Array: computationally efficient list of numbers
- Arrays have fixed length and one numeric type
- Arrays require the `numpy` module

Examples on using arrays

```
>>> import numpy
>>> L = [1, 4, 10.0] # List of numbers
>>> a = numpy.array(L) # Convert to array
>>> print(a)
[ 1. 4. 10.]
>>> print(a[1]) # indexing
4.0
>>> print(a[0:2]) # slice (index 2 not included!)
[1. 4.]
>>> print(a.dtype) # data type of an element
float64
>>> b = 2*a + 1 # arithmetics on arrays
>>> print(b)
[ 3. 9. 21.]
```

numpy functions create entire arrays

Apply ln to all elements in array `a`:

```
>>> c = numpy.log(a)
>>> print(c)
[0. 1.38629436 2.30258509]
```

Create $n+1$ uniformly spaced coordinates in $[a, b]$:

```
t = numpy.linspace(a, b, n+1)
```

Array of length n filled with zeros:

```
t = numpy.zeros(n)
s = numpy.zeros_like(t) # zeros with t's size and dtype
```

Let's use arrays in our previous program

```
import numpy as np

v0 = 2
a = 0.2
dt = 0.1 # Increment
n = int(round(2 / dt)) + 1 # No of t values

t_values = np.linspace(0, 2, n)
s_values = v0 * t_values + 0.5 * a * t_values**2

# Make nicely formatted table
for t, s in zip(t_values, s_values):
    print(f'{t:.2f} {s:.4f}')
```

Note: no explicit loop for computing `s_values`!

Standard math functions: the `math` module

```
>>> import math
>>> print(math.sin(math.pi))
1.2246467991473532e-16 # approximate value
```

Get rid of the `math` prefix:

```
from math import sin, pi
print(sin(pi))

from math import * # import everything
print(sin(pi), log(e), tanh(0.5))
```

Use numpy for math on arrays

Matlab-style:

```
from numpy import *
x = linspace(0, 1, 101)
y = exp(-x)*sin(pi*x)
```

Preferred style:

```
import numpy as np
x = np.linspace(0, 1, 101)
y = np.exp(-x)*np.sin(np.pi*x)
```

Our convention: `np` prefix, clean formulas

```
import numpy as np
x = np.linspace(0, 1, 101)

from numpy import sin, exp, pi
y = exp(-x)*sin(pi*x)
```

Array assignment gives a view (no copy!)

Consider array assignment `b = a`:

```
a = np.linspace(1, 5, 5)
b = a
```

Here, `b` is a *view* (pointer) to the data of `a` — no copying of data!

```
>>> a
array([1., 2., 3., 4., 5.])
>>> b[0] = 5 # changes a[0] to 5
>>> a
array([5., 2., 3., 4., 5.])
>>> a[1] = 9 # changes b[1] to 9
>>> b
array([5., 9., 3., 4., 5.])
```

Copying array data: the copy method

```
>>> c = a.copy() # copy all elements to new array c
>>> c[0] = 6 # a is not changed
>>> a
array([1., 2., 3., 4., 5.])
>>> c
array([6., 2., 3., 4., 5.])
```

Construction of tridiagonal and sparse matrices

- SciPy offers scipy.sparse
- Build sparse matrices from diagonals with `diags` (or the newer `diags_array`)
- Super-/sub-diagonals are shorter than the main diagonal
- Scalar broadcasting is supported (specify `shape`)

Tridiagonal matrix with `diags`

```
>>> import numpy as np
>>> import scipy.sparse
>>> N = 6
>>> diagonals = [
... -np.linspace(1, N, N)[:-1],
... -2*np.ones(N),
... np.linspace(1, N, N)[1:],
... ]
>>> A = scipy.sparse.diags(diagonals, [-1, 0, 1], format='csc')
>>> A.toarray()
[[-2. 2. 0. 0. 0. 0.]
 [-1. -2. 3. 0. 0. 0.]
 [ 0. -2. -2. 4. 0. 0.]
 [ 0. 0. -3. -2. 5. 0.]
 [ 0. 0. 0. -4. -2. 6.]
 [ 0. 0. 0. 0. -5. -2.]]
```

Scalar broadcasting with `diags`

```
>>> B = scipy.sparse.diags(
... [1, 2, 3], [-2, 0, 1], shape=(6, 6), format='csc')
>>> B.toarray()
[[2. 3. 0. 0. 0. 0.]
 [0. 2. 3. 0. 0. 0.]
 [1. 0. 2. 3. 0. 0.]
 [0. 1. 0. 2. 3. 0.]
 [0. 0. 1. 0. 2. 3.]
 [0. 0. 0. 1. 0. 2.]]
```

Legacy note: `spdiags`

Older code may use `spdiags`: all diagonals must have length N; unused entries sit at the start of super-diagonals and the end of sub-diagonals.

```
>>> diagonals = np.zeros((3, N))
>>> diagonals[0, :] = np.linspace(-1, -N, N)
>>> diagonals[1, :] = -2
>>> diagonals[2, :] = np.linspace(1, N, N)
>>> A = scipy.sparse.spdiags(
... diagonals, [-1, 0, 1], N, N, format='csc')
```

Prefer `diags` / `diags_array` in new code.

Solving a tridiagonal system

Solve $Ax = b$: choose x, compute $b = Ax$, solve, check.

```
>>> x = np.linspace(-1, 1, N)
>>> b = A @ x # sparse matrix-vector product
>>> import scipy.sparse.linalg
>>> x = scipy.sparse.linalg.spsolve(A, b)
>>> print(x)
[-1. -0.6 -0.2 0.2 0.6 1. ]
```

Check against dense matrix computations

```
>>> A_d = A.toarray()
>>> b = A_d @ x # dense product
>>> x = np.linalg.solve(A_d, b)
>>> print(x)
[-1. -0.6 -0.2 0.2 0.6 1. ]
```

See also `scipy_intro/tridiagonal.py` in this repository.

Plotting

Plotting is done with `matplotlib`:

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


def main():
    v0 = 0.2
    a = 2
    n = 21 # No of t values for plotting

    t = np.linspace(0, 2, n + 1)
    s = v0 * t + 0.5 * a * t**2

    plt.plot(t, s)
    plt.xlabel('t [s]')
    plt.ylabel('s [m]')
    plt.savefig('myplot.png')
    plt.show()


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

Plotting of multiple curves

Core idea (see `python_intro/plot2.py`):

```
v0, a0, a1, n = 0.2, 2, 3, 21
t = np.linspace(0, 2, n + 1)
s0 = v0 * t + 0.5 * a0 * t**2
s1 = v0 * t + 0.5 * a1 * t**2
plt.plot(t, s0, 'r-', t, s1, 'bo')
plt.xlabel('t [s]'); plt.ylabel('s [m]')
plt.legend([rf'$s(t; v_0={v0}, a={a0})$',
            rf'$s(t; v_0={v0}, a={a1})$'])
```

Outline

Functions

- $s(t) = v_0 t + \frac{1}{2} a t^2$ is a mathematical function
- Can implement $s(t)$ as a Python function `s(t)`

```
def s(t):
    return v0*t + 0.5*a*t**2

v0 = 0.2
a = 4
value = s(3) # Call the function
```

Function notes

- functions start with the keyword `def`
- statements belonging to the function must be indented
- function input is represented by arguments (comma-separated if more than one)
- function output is returned to the calling code
- `v0` and `a` above are *global variables*, which must be initialized before `s(t)` is called

Functions can have multiple arguments

`v0` and `a` as function arguments instead of globals:

```
def s(t, v0, a):
    return v0*t + 0.5*a*t**2

value = s(3, 0.2, 4) # Call the function

# More readable call
value = s(t=3, v0=0.2, a=4)
```

Keyword arguments (default values)

```
def s(t, v0=1, a=1):
    return v0*t + 0.5*a*t**2

value = s(3, 0.2, 4) # specify new v0 and a
value = s(3) # rely on v0=1 and a=1
value = s(3, a=2) # rely on v0=1
value = s(3, v0=2) # rely on a=1
value = s(t=3, v0=2, a=2) # specify everything
value = s(a=2, t=3, v0=2) # any keyword order
```

Positional vs keyword arguments

- Arguments without the argument name are *positional*
- Positional arguments must always be listed before keyword arguments in the definition and in any call
- The sequence of keyword arguments can be arbitrary

Vectorization speeds up the code

Scalar code (one number at a time):

```
def s(t, v0, a):
    return v0*t + 0.5*a*t**2

for i in range(len(t_values)):
    s_values[i] = s(t_values[i], v0, a)
```

Vectorized code: apply `s` to the entire array

```
s_values = s(t_values, v0, a)
```

How can vectorization work?

Expression `v0*t + 0.5*a*t**2` with array `t`:

- `r1 = v0*t` (scalar times array)
- `r2 = t**2` (square each element)
- `r3 = 0.5*a*r2` (scalar times array)
- `r1 + r3` (add each element)

Scalar functions often work for arrays too

True if computations involve arithmetic and math functions:

```
from math import exp, sin

def f(x):
    return 2*x + x**2*exp(-x)*sin(x)

v = f(4) # works with scalar x

from numpy import exp, sin, linspace
x = linspace(0, 4, 100001)
v = f(x) # now works with array x
```

Functions can return multiple values

Return $s(t) = v_0 t + \frac{1}{2} a t^2$ and $s'(t) = v_0 + at$:

```
def movement(t, v0, a):
    s = v0*t + 0.5*a*t**2
    v = v0 + a*t
    return s, v

s_value, v_value = movement(t=0.2, v0=2, a=4)
```

Multiple return values are a tuple

`return s, v` returns a *tuple* (≈ immutable list):

```
>>> def f(x):
... return x+1, x+2, x+3
...
>>> r = f(3)
>>> print(r)
(4, 5, 6)
>>> type(r)
<class 'tuple'>
>>> print(r[1])
5
```

Tuples are constant lists (cannot be changed).

A more general formula (part I)

Equations from basic kinematics:

$$v = \frac{ds}{dt}, \quad s(0) = s_0$$
$$a = \frac{dv}{dt}, \quad v(0) = v_0$$

Integrate to find $v(t)$:

$$\int_0^t a(t)\,dt = \int_0^t \frac{dv}{dt}\,dt$$

which gives

$$v(t) = v_0 + \int_0^t a(t)\,dt$$

A more general formula (part II)

Integrate again over $[0, t]$ to find $s(t)$:

$$s(t) = s_0 + v_0 t + \int_0^t \left(\int_0^\tau a(\xi)\, d\xi \right) d\tau$$

Example: $a(t) = a_0$ for $t \in [0, t_1]$, then $a(t) = 0$ for $t > t_1$:

$$s(t) = \begin{cases} s_0 + v_0 t + \frac{1}{2} a_0 t^2, & t \le t_1 \\ s_0 + v_0 t_1 + \frac{1}{2} a_0 t_1^2 + a_0 t_1 (t - t_1), & t > t_1 \end{cases}$$

Need an *if* test to implement this!

Basic if-else tests

```
if condition:
    <statements when condition is True>
else:
    <statements when condition is False>
```

Here, `condition` is a boolean expression with value `True` or `False`.

```
if t <= t1:
    s = v0*t + 0.5*a0*t**2
else:
    s = v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1)
```

Multi-branch if tests

```
if condition1:
    <statements when condition1 is True>
elif condition2:
    <when condition1 is False and condition2 is True>
elif condition3:
    <when 1 and 2 are False and condition3 is True>
else:
    <when condition1/2/3 all are False>
```

Just if, no else:

```
if condition:
    <statements when condition is True>
```

Piecewise function with if

```
def s_func(t, v0, a0, t1):
    if t <= t1:
        s = v0 * t + 0.5 * a0 * t**2
    else:
        s = (v0 * t + 0.5 * a0 * t1**2
             + a0 * t1 * (t - t1))
    return s
```

Full demo: `python_intro/plot3.py`

If + arrays: does not work

```
>>> def f(x):
... return x if x < 1 else 2*x
...
>>> import numpy as np
>>> x = np.linspace(0, 2, 5)
>>> f(x)
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

Problem: `x < 1` evaluates to a boolean array, not a single boolean.

Remedy 1: call with scalar arguments

```
n = 201
t1 = 1.5
v0 = 0.2
a0 = 20
t = np.linspace(0, 2, n + 1)
s = np.zeros(n + 1)
for i in range(len(t)):
    s[i] = s_func(t=t[i], v0=v0, a0=a0, t1=t1)

plt.plot(t, s, 'b-')
plt.plot([t1, t1], [0, s_func(t=t1, v0=v0, a0=a0, t1=t1)], 'r--')
```

Remedy 2: vectorize with `where`

```
s = np.where(condition, s1, s2)
```

- `condition`: array of boolean values
- `s[i] = s1[i]` if `condition[i]` is `True`
- `s[i] = s2[i]` if `condition[i]` is `False`

```
s = np.where(t <= t1,
             v0*t + 0.5*a0*t**2,
             v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1))
```

Remedy 3: vectorize with array indexing

- Let `b` be a boolean array (e.g. `b = t <= t1`)
- `s[b]` selects elements where `b[i]` is `True`
- Assign: `s[b] = (expr)[b]`

```
s = np.zeros_like(t)
s[t <= t1] = (v0*t + 0.5*a0*t**2)[t <= t1]
s[t > t1] = (v0*t + 0.5*a0*t1**2
             + a0*t1*(t-t1))[t > t1]
```

See also `python_intro/plot3_vec.py`.

Outline

File reading

Put input data in a text / YAML file (modern approach in this repo):

```
# input.yaml
v0: 2.0
a: 0.2
dt: 0.1
interval: [0, 2]
```

Question

How can we read this file into variables `v0`, `a`, `dt`, and `interval`?

Reading parameters with YAML

```
import numpy as np
import yaml

with open('input.yaml') as infile:
    params = yaml.safe_load(infile)

v0 = params['v0']
a = params['a']
dt = params['dt']
interval = params['interval']

t_values = []
s_values = []
n = int(round(interval[1] / dt)) + 1 # No of t values
for i in range(n):
    t = i * dt
    s = v0 * t + 0.5 * a * t**2
    t_values.append(t)
    s_values.append(s)
```

Classic line parsing: `variable = value`

Older style (still useful to understand):

```
with open('input.dat') as infile:
    for line in infile:
        name, value = line.split('=')
        name = name.strip()
        if name in ('v0', 'a', 'dt'):
            exec(f'{name} = float({value})')
        elif name == 'interval':
            interval = eval(value)
```

Splitting lines into words

```
>>> line = 'v0 = 5.3'
>>> variable, value = line.split('=')
>>> variable
'v0 '
>>> value
' 5.3'
>>> variable.strip() # strip away blanks
'v0'
```

Note: must convert `value` to `float` before computing!

The magic `eval` function

`eval(s)` executes a string `s` as a Python expression and creates the corresponding object:

```
>>> obj1 = eval('1+2')
>>> obj1, type(obj1)
(3, <class 'int'>)
>>> obj2 = eval('[-1, 8, 10, 11]')
>>> obj2, type(obj2)
([-1, 8, 10, 11], <class 'list'>)
>>> from math import sin, pi
>>> x = 1
>>> obj3 = eval('sin(pi*x)')
>>> obj3, type(obj3)
(1.2246467991473532e-16, <class 'float'>)
```

Why is `eval` useful?

We can read formulas, lists, expressions as text from a file and with `eval` turn them into live Python objects.

Caution

Only use `eval` on trusted input (never on arbitrary user strings from the web).

Implementing a calculator in Python

Demo:

```
Terminal> uv run python calc.py "1 + 0.5*2"
2.0
Terminal> uv run python calc.py "sin(pi*2.5) + exp(-4)"
1.0183156388887342
```

Just a few lines:

```
import sys
from math import * # sin, cos, exp, pi, etc.
expression = sys.argv[1]
result = eval(expression)
print(result)
```

The `with` statement for file handling

```
with open('input.dat', 'r') as infile:
    for line in infile:
        ...
```

No need to close the file when using `with`.

File writing

- We have t and $s(t)$ in lists `t_values`, `s_values`
- Task: write a nicely formatted table to a file

```
with open('table1.dat', 'w') as outfile:
    outfile.write('# t s(t)\n')
    for t, s in zip(t_values, s_values):
        outfile.write(f'{t:.2f} {s:.4f}\n')
```

Simplified writing: `numpy.savetxt`

```
data = np.array([t_values, s_values]).transpose()
np.savetxt('table2.dat', data, fmt=['%.2f', '%.4f'],
           header='t s(t)', comments='# ')
```

Resulting `table2.dat`:

```
# t s(t)
0.00 0.0000
0.10 0.2010
0.20 0.4040
...
2.00 4.4000
```

Simplified reading: `numpy.loadtxt`

```
data = np.loadtxt('table2.dat', comments='#')
```

Note:

- Lines beginning with `#` are skipped
- `data` is a 2-D array: `data[i,0]` holds t and `data[i,1]` holds $s(t)$ in row i

Full script: `python_intro/for_file_read_input.py`

Outline

Why classes?

- All objects in Python are made from a class
- You do not need to know about classes to use Python
- But class programming is powerful

- Class = functions + variables packed together
- A class is a logical unit in a program
- A large program as a combination of appropriate units

A very simple class

- One variable: `a`
- One function: `dump` for printing `a`

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

    def dump(self):
        print(self.a)
```

Terminology: functions are called *methods* and variables are called *attributes*.

How can we use this class?

First, make an *instance* (object) of the class:

```
t = Trivial(a=4)
t.dump()
```

Note:

- `Trivial(a=4)` means `Trivial.__init__(t, 4)`
- `self` is an argument in `__init__` and `dump`, but not used in the calls
- `__init__` is the *constructor*
- `t.dump()` means `Trivial.dump(t)` (`self` is `t`)

The `self` argument

It takes time and experience to understand `self`!

1. `self` must always be the first argument
2. `self` is never used in calls
3. `self` is used to access attributes and methods inside methods

`self` is confusing at first, but later it greatly helps understanding how classes work!

A class for a mathematical function

Function with one independent variable t and parameters v_0, a:

$$s(t; v_0, a) = v_0 t + \frac{1}{2} a t^2$$

Class representation:

- `v0` and `a` are attributes (data)
- A method to evaluate $s(t)$ as a function of `t` only

Usage:

```
s = Distance(v0=2, a=0.5) # create instance
v = s(t=0.2) # compute formula
```

The class code

```
class Distance:
    def __init__(self, v0, a):
        self.v0 = v0
        self.a = a

    def __call__(self, t):
        v0, a = self.v0, self.a
        return v0*t + 0.5*a*t**2

s = Distance(v0=2, a=0.5)
v = s(t=0.2) # actually s.__call__(t=0.2)
```

General $f(x, y, z; p_1, \ldots, p_n)$

- The n parameters $p_1, \ldots, p_n$ are attributes
- `__call__(self, x, y, z)` computes $f(x, y, z)$

```
class F:
    def __init__(self, p1, p2, ...):
        self.p1 = p1
        self.p2 = p2
        ...

    def __call__(self, x, y, z):
        # return formula with x, y, z and self.p1, ...

f = F(p1=..., p2=..., ...)
print(f(1, 4, -1))
```