A very basic introduction to scientific Python programming

Hans Petter Langtangen[1,2] (`hpl@simula.no`)

[1]Simula Research Laboratory
[2]University of Oslo

Jan 8, 2015

Contents.

This note introduces very basic programming elements, such as

- variables for numbers, lists, and arrays

- while loops and for loops

- functions

- if tests

- plotting

through examples involving a mathematical formula. A glimpse of vectorization of code is also given.

Contents

1 Variables, loops, lists, and arrays

1.1 Getting access to Python

Simple mathematical calculations can be done in plain Python, but for more advanced scientific computing, as we do here, several add-on packages are needed. Getting all software correctly installed used to be quite challenging, but today there are several easy-to-use approaches to get access to Python.

Mac and Windows. We recommend to download and install Anaconda[1], which is a free distribution of Python that comes with most of the packages you need for advanced scientific computing.

Ubuntu. Debian-based Linux systems, such as Ubuntu, can also use Anaconda, but it is more common to use the `apt-get` command-line tool or the Ubuntu installer to install a set of Debian packages. Here is a list of the packages you need to install for this introduction:

```
Terminal> sudo apt-get install python-pip python-setuptools \
  python-scipy python-sympy python-cython python-matplotlib \
  python-dev python-profiler pydb spyder imagemagick gedit vim \
  emacs python-mode git mercurial lib-avtools gnome-terminal
```

In addition, run

```
Terminal> pip install nose
Terminal> pip install pytest
Terminal> pip install ipython --upgrade
Terminal> pip install tornado --upgrade
Terminal> pip install pyzmq --upgrade
```

Web access. You can also access Python directly through a web browser without having it installed on your local computer. We refer to the document How to access Python for doing scientific computing[2] for more details on such tools and also installations based on Anaconda and Ubuntu.

---

[1] `https://store.continuum.io/cshop/anaconda/`
[2] `http://hplgit.github.io/edu/accesspy/accesspy.html`

1.2   Mathematical example

We shall use a famous mathematical formula in our forthcoming programming examples. The formula tells how long distance s an object has moved in a time interval $[0, t]$ if it starts out with a velocity v_0 and undergoes constant acceleration a:

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

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

1.3   A program for evaluating a formula

Here is a Python program for computing s, given $t = 0.5$, $v_0 = 2$, and $a = 0.2$:

```
t = 0.5
v0 = 2
a = 0.2
s = v0*t + 0.5*a*t**2
print s
```

The program is pure text and must be placed in a *pure text file* using a *text editor*. Popular text editors are Gedit, Nano, Emacs, and Vim on Linux, TextWrangler on Mac OS X, and Notepad++ on Windows. Save the text to a program file whose name ends in `.py`, say `distance.py`.

   The program is run in a terminal window (Command Prompt on Windows, Terminal application on Mac OS X, `xterm` or `gnome-terminal` on Linux):

```
Terminal> python distance.py
1.025
```

The result of the `print` statement is the number `1.025` in the terminal window.

   As an alternative to writing programs in a text editor and executing them in a terminal window, you may use the Spyder[3] graphical interface which gives a more Matlab-style environment to work in. Anaconda installations come with Spyder.

   The `distance.py` program first contains four *assignment statements* where we assign numbers or the results of arithmetic expressions to *variables*. The variables have names coinciding with the mathematical notation used in the formula: `t`, `v0`, `a`, and `s`. You may think of variables in this programs just as variables in mathematics.

More technical details.

A statement like `t = 0.5` works as follows. First, the right-hand side is interpreted by Python as a real number and a `float` object containing

---

[3] `https://code.google.com/p/spyderlib/`

the value 0.5 is created. Then the name `t` is defined as a reference to this object.

In the statement `s = v0*t + 0.5*a*t**2`, Python will first go to the right-hand side and observe that it is an arithmetic expression involving variables with known values (or names referring to existing objects). The arithmetic expression is calculated, resulting in a real number that is saved as a `float` object with value 1.025 in the computer's memory.

Everything in Python is an object of some type. Here, `t`, `a`, and `s` are `float` objects, representing real (floating-point) numbers, while `v0` is an `int` object, representing the integer 2. There are many other types of objects: strings, lists, tuples, dictionaries, arrays, files, ...

1.4 Formatted output with text and numbers

For any object `s` in Python, `print s` will (normally) print the contents of `s`. However, sometimes we want to combine the content of an object with some text. Say we want to print `s=1.025` rather than just the number. This is easily accomplished using *printf* syntax:

```
print 's=%g' % s
```

The output is specified as a string, enclosed in single or double quotes. Inside the string, there can be "slots" for numbers (or other objects), indicated by a percentage sign followed by a specification of kind of data that will be inserted at this place. In the string above, there is one such slot, `%g`, where the `g` implies a real number written as compactly as possible.

It is easy to control the number of decimals using printf syntax. Printing out `s=1.03`, i.e., `s` with two decimals, is done by

```
print 's=%.2f' % s
```

where the `f` signifies a *decimal number* and the preceding `.2` means 2 decimals. Scientific notation, as in `s=1.03E+00` ($1.03 \cdot 10^0$), is specified as `%.2E` (2 decimals).

The printf syntax is available in numerous programming languages. Python also offers a related variant, called *format string syntax*:

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

1.5 While loops

Suppose we want to make a table with two columns, one with t values and one with the corresponding s values. Now we have to repeat a lot of calculations with the formula (1). This is easily done with a *loop*. There are two types of loops in Python: *while* loops and *for* loops.

Let the t values go from 0 to 2 in increments of 0.1. The following program applies a *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
```

The result of running this program in a terminal window is

```
Terminal> 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
0.6 1.236
0.7 1.449
0.8 1.664
0.9 1.881
1.0 2.1
1.1 2.321
1.2 2.544
1.3 2.769
1.4 2.996
1.5 3.225
1.6 3.456
1.7 3.689
1.8 3.924
1.9 4.161
```

So, how do we interpret the contents of this program? First we initialize four variables: `v0`, `a`, `dt`, and `t`. Everything after `#` on a line is a *comment* and does not affect what happens in the program, but is meant to be of help for a human reading the program. Then comes the while loop:

```
while condition:
    <intented statement>
    <intented statement>
    <intented statement>
```

Observe the colon at the end of the `while` line. The set of indented statements are repeated as long as the expression or variable `condition` evaluates to `True`. In the present case, the condition is the *boolean expression* `t <= 2`, so as long as the value is less than or equal to 2, `t <= 2` evaluates to `True`, otherwise it evaluates to `False`.

The Python Online Tutor[4] is a great tool to examine the program flow. Consider this little loop run in the Python Online Tutor:

```
a = 1
da = 0.5
while a <= 3:
    print a
    a = a + da
```

[4]http://pythontutor.com/visualize.html#mode=edit

In the Python Online Tutor you can execute each statement, follow the program flow, and always see the contents of the various variables.

1.6 Lists

The table created in the previous section has two columns of data. We could store all the numbers in each column in a *list* object. A list is just a collection of objects, here numbers, in a given sequence. For example,

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

is a list of three numbers. Lists are encolsed in square brackets and may contain any type of objects separated by commas. Here we mix a filename (string), a real number, and an integer:

```
L = ['mydata.txt', 3.14, 10]
```

The different list *elements* can be reached via indexing: `L[0]` is the first element, `L[1]` is the second, and `L[-1]` is the last element. Here is an *interactive Python shell* where we can write Python statements and examine the contents of variables as we perform various operations:

```
>>> L = ['mydata.txt', 3.14, 10]
>>> print L[0]
mydata.txt
>>> print L[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 of the list
>>> print L
[3.14, 10, -1]
```

Let us store the numbers in the previous table in lists, one for each column. We can start with empty lists `[]` and use `append` to add a new element to the lists in each pass of the while loop. Thereafter, we can run a new while loop and print the contents of the lists in a nice, tabular fashion:

```
v0 = 2
a = 0.2
dt = 0.1   # Increment
t = 0
t_values = []
s_values = []
while t <= 2:
    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 a nicely formatted table
```

```
    i = 0
    while i <= len(t_values)-1:
        print '%.2f  %.4f' % (t_values[i], s_values[i])
        i += 1    # Same as i = i + 1
```

The output looks like

```
[0.0, 0.201, 0.404, 0.6090000000000001, 0.8160000000000001,
 1.025, 1.236, 1.4489999999999998, 1.664, 1.8809999999999998,
 2.0999999999999996, 2.3209999999999997, ...]
0.00  0.0000
0.10  0.2010
0.20  0.4040
0.30  0.6090
0.40  0.8160
0.50  1.0250
0.60  1.2360
0.70  1.4490
...
```

Note that `print s_values` here leads to output with many decimals and small round-off errors. To get complete control of the formatting of real numbers in the table, we use the printf syntax.

Lists come with a lot of functionality. See the Python Tutorial[5] for many more examples.

1.7 For loops

A for loop is used for visiting elements in a list, one by one:

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

The variable `e` is successively set equal to the elements in the list, in the right order. Note the colon at the end of the `for` line. The statements to be executed in each pass of the loop (here only `print e`) must be indented. When `e` is set equal to the last element and all indented statements are executed, the loop is over, and the program flow continues with the next statement that is not indented. Try the following code out in the Python Online Tutor:

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

A for loop over the valid indices in a list is created by

---

[5]`https://docs.python.org/2/tutorial/introduction.html#lists`

```
for i in range(len(somelist)):
    # Work with somelist[i]
```

The `range` function returns a list of integers: `range(a, b, s)` returns the integers `a`, `a+s`, `a+2*s`, ... up to *but not including* `b`. Just writing `range(b)` implies `a=0` and `s=1`, so `range(len(somelist))` returns `[0, 1, 2]`.

For loops over real numbers.

The `for i in range(...)` construction can only run a loop over integers. If you need a loop over real values, you have to either create a list with the values first, or use a formula where each value is generated through an integer counter:

```
# Need loop over 0, 0.1, 0.2, ..., 1
values = []
for i in range(11):
    values.append(i*0.1)

# Shorter version using list comprehension (same as the loop above
values = [i*0.1 for i in range(11)]

for value in values:
    # work with value
```

We can now rewrite our program that used lists and while loops to use for loops instead:

```
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 '%.2f  %.4f' % (t, s)

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

Observe that we have here used a slightly different technique for computing the t values inside the first loop: now we set t as $i\Delta t$, where Δt (`dt` in the code) is the increment (0.1) between each t value. The computation of `n`, the number of t values, makes use of `round` to make a correct mathematical rounding to the nearest integer (and `int` makes an integer object out of the rounded real

number). (In an interval $[a, b]$ divided into subintervals of equal length Δt, there will be $1 + (b - a)/\Delta t$ points in total.)

Running through multiple lists simultaneously is done with the `zip` construction:

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

One may instead create a for loop over all the legal index values instead and index each array,

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

1.8   Arrays

Lists are useful for collecting a set of numbers or other objects in a single variable. Arrays are much like lists, but tailored for collection of numbers. The primary advantage of arrays is that you can use them very efficiently and conveniently in mathematical computations, but the downside is that an array has (in practice) a fixed length and all elements must be of the same type. This is usually no important restriction in scientific computations.

To use arrays, you must import the `numpy` module:

```
>>> import numpy
>>> L = [1, 4, 10.0]      # List of numbers
>>> a = numpy.array(L)    # Make corresponding array
>>> print a
[  1.   4.   10.]
>>> print a[1]
4.0
>>> print a.dtype         # Data type of an element
float64
>>> b = 2*a + 1
>>> print b
[  3.   9.   21.]
```

Note that all elements in the `a` array are of `float` type (because one element in L was `float`). Arithmetic expressions such as `2*a+1` work with `a` as array, but not as list. In fact, we can pass arrays to mathematical functions:

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

The `numpy` module has a lot of very useful utilities. To create $n + 1$ uniformly distributed coordinates in an interval $[a, b]$, stored in an array, one can use `linspace`:

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

This construction makes it easy to create arrays for the t and s values in our tables:

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

t_values = numpy.linspace(0, 2, n+1)
s_values = v0*t + 0.5*a*t**2

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

1.9 Mathematical functions

Python offers access to all standard mathematical functions such as $\sin x$, $\cos x$, $\tan x$, $\sinh x$, $\cosh x$, $\tanh x$, all their inverses (called `asin(x)`, `asinh(x)`, and so forth), e^x (`exp(x)`), $\ln x$ (`log(x)`), and $x!$ (`factorial(x)`). However, one has to import a module to get access to these functions. For scalars (single numbers) the relevant module is `math`:

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

which shows that the sine function is only approximate (to 16 digits). Many prefer to write mathematical expressions without the `math` prefix:

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

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

The `numpy` module contains sine, cosine, and other mathematical functions that work on scalars as well as arrays.

Import of `numpy`.

To get Python code that is as similar to Matlab as possible, one would do

```
from numpy import *
```

However, in the Python community it has been a culture to use a prefix `np` as abbreviation for `numpy`:

```
import numpy as np
print np.sin(np.pi)
```

Our convention is to use the **np** prefix for typical Matlab functions, but skip the prefix when working with mathematical functions like the sine and

---

cosine to get a one-to-one correspondence between formulas in the program and in the mathematical description of the problem.

```python
import numpy as np
from numpy import sin, exp
t = np.linspace(0, 4, 1001)
p = exp(-t)*sin(2*t)
```

1.10   Plotting

We can easily make a graph of a function $s(t)$ using the module `matplotlib`. The technique is to compute an array of t values and a corresponding array of function values $s(t)$. Plotting programs will draw straight lines between the points on the curve, so a sufficient number of points are needed to give the impression of a smooth curve. Our $s(t)$ function is plotted by the following code:
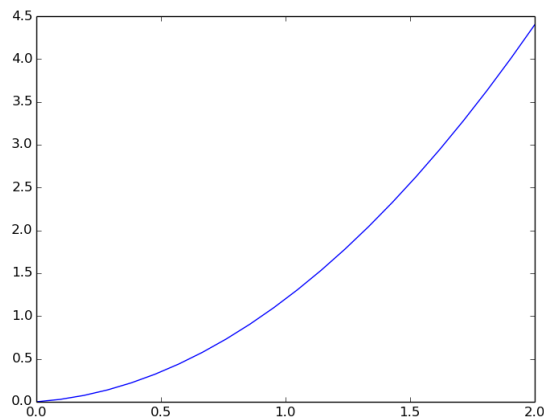
```python
import numpy as np
import matplotlib.pyplot as plt

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.savefig('myplot.png')
plt.show()
```

The plotfile `myplot.png` looks like

Matlab users may prefer to do

```python
from numpy import *
from matplotlib.pyplot import *
```

11

such that they can use `linspace` and `plot` without any prefix, just as in Matlab.

Two curves can easily be plotted, this time also with labels on the axis and a box with legends such that we can distinguish the two curves:

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

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.savefig('myplot.png')
plt.show()
```
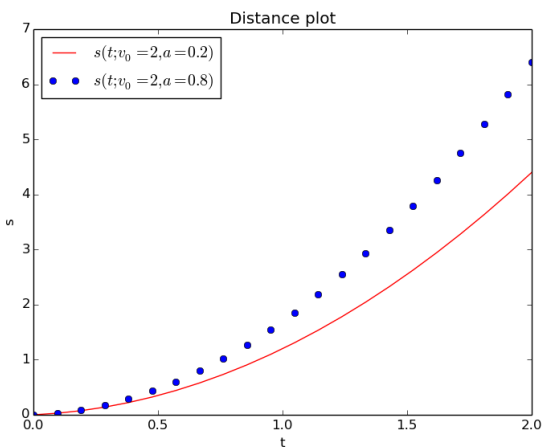

2 Functions and branching

2.1 Functions

Since $s(t)$ is a function in mathematics, it is convenient to have it as a function in Python too:

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

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

Functions start with the keyword `def`, and all the statements belonging to the function must be indented.

In this program, `v0` and `a` are *global variables*, which must be defined before calling the `s(t)` function, otherwise `v0` and `a` becomes undefined variables in the expression inside the function.

Instead of having `v0` and `a` as global variables, we may let them be function arguments:

```python
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)
```

Vectorized functions.

Applying the function `s(t, v0, a)` to an array `t` can be done in two ways:

```python
# Scalar code: work with one element at a time
for i in range(len(t)):
    s_values[i] = s(t_values[i], v0, a)

# Vectorized code: apply s to the whole array
s_values = s(t_values, v0, a)
```

For the last line to work, the function `s` must contain statements that work correctly when `t` is an array argument.

The advantage of vectorized code is that we avoid a loop in Python. Instead, we carry out mathematical operations on entire arrays, e.g., `v0*t` and `a*t**2`. Technically, such (binary) operations are executed as loops in very fast (compiled) C code.

A function can return more than one value, say $s(t)$ and the velocity $s'(t) = v_0 + at$:

```python
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)
```

When n values are returned, we list n variables on the left-hand side in the call.

Python functions return only one object.

Even when we return several values, as in `return s, v`, actually only one object is returned. The `s` and `v` values are packed together in a *tuple* object (which is very similar to a list).

```
>>> def f(x):
...     return x+1, x+2, x+3
...
>>> r = f(3)       # Store all three return values in one object r
>>> print r
(4, 5, 6)
>>> type(r)        # What type of object is r?
<type 'tuple'>
>>> print r[1]
5
```

Tuples are constant lists, so you can index them as lists, but you cannot
change the contents (`append` or `del` is illegal).

2.2 A more general mathematical formula

The formula (1) arises from the basic differential equations in kinematics:

$$v = \frac{ds}{dt}, \quad s(0) = s_0, \tag{2}$$

$$a = \frac{dv}{dt}, \quad v(0) = v_0. \tag{3}$$

Given any acceleration $a(t)$, we can solve for $s(t)$ through integration. First, we
integrate to find $v(t)$:

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

which gives

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

Then we integrate again over $[0,t]$ to find $s(t)$:

$$s(t) = s_0 + v_0 t + \int_0^t \left(\int_0^t a(t)dt \right) dt.$$

Suppose we have some constant acceleration a_0 in $[0, t_1]$ and no acceleration
thereafter. We find

$$s(t) = \begin{cases} s_0 + v_0 t + \frac{1}{2}a_0 t^2, & t \le t_1 \\ s_0 + v_0 t_1 + \frac{1}{2}a_0 t_1^2 + a_0 t_1(t - t_1), & t > t_1 \end{cases} \tag{4}$$

To implement a function like (4), we need to branch into one type of code
(formula) if $t \le t_1$ and another type of code (formula) if $t > t_1$. This is called
branching and the *if test* is the primary construction to use.

14

2.3   If tests

An if test has the structure

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

Here, `condition` is a boolean expression that evaluates to `True` or `False`. For the picewisely defined function (4) we would use an if test in the implementation:

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

The `else` part can be omitted when not needed. Several branches are also possible:

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

A Python function implementing the mathematical function (4) reads

```
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
```

To plot this $s(t)$, we need to compute points on the curve. Trying to call `s_func` with an array `t` as argument will not work because of the if test. In general, functions if if tests will not automatically work with arrays. One solution is to compute the function values one by one in a loop. Appropriate code is

```
n = 201   # No of t values for plotting
t1 = 1.5

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=0.2, a0=20, t1=t1)
```

Relevant statements for plotting are now

```
plt.plot(t, s, 'b-')
plt.plot([t1, t1], [0, s_func(t=t1, v0=0.2, a0=20, t1=t1)], 'r--')
plt.xlabel('t')
plt.ylabel('s')
plt.savefig('myplot.png')
plt.show()
```

15

> **Vectorization of functions with if tests.**
>
> To vectorize the computation of the array of $s(t)$ values, i.e., avoid a loop where `s_func` is called for each `t[i]` element, we must completely rewrite the if test. There are two methods: the `numpy.where` function and array indexing.
>
> **Using the `where` function.** A vectorized if-else test can be coded as
>
> ```
> s = np.where(condition, s1, s2)
> ```
>
> Here, `condition` i[i]'s an array of boolean values, and `s[i] = s1[i]` if `condition[i]` is `True`, and otherwise `s[i] = s2[i]`.
>   Our example then becomes
>
> ```
> s = np.where(t <= t1,
>              v0*t + 0.5*a0*t**2,
>              v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1))
> ```
>
> Note that `t <= t1` with array `t` and scalar `t1` results in a boolean array `b` where `b[i] = t[i] <= t1`.
>
> **Using array indexing.** It is possible to index a subset of indices in an array `s` using a boolean array `b`: `s[b]`. This construction picks out all the elements `s[i]` where `b[i]` is to `True`. On the right-hand side we can then assign some array expression `expr` of the same length as `s[b]`:
>
> ```
> s[b] = (expr)[b]
> ```
>
> Our example can utilize this technique with `b` as `t <= t1` and `t > t1`:
>
> ```
> s = np.zeros_like(t)   # Make s as zeros, same size & type as 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]
> ```

3   Exercises

Exercise 1: Program a formula

Make a Python program that calculates and prints the value of the formula

$$y = 6x^2 + 3x + 2,$$

for $x = 2$.

Exercise 2: Combine text and numbers in output

Let $y = x^2$. Make a program that writes the text

```
y(2.550)=6.502
```

if $x = 2.55$. The values of x and y should be written with three decimals. Run the program for $x = \pi$ (the value if π is available as the variable `pi` in the `math` module).

Exercise 3: Program a while loop

Define a sequence of numbers,

$$x_n = n^2 + 1,$$

for integers $n = 0, 1, 2, \ldots, N$. Write a program that prints out x_n for $n = 0, 1, \ldots, 20$ using a while loop.

Exercise 4: Create a list with a while loop

Store all the x_n values computed in Exercise 3 in a list (using a while loop). Print the entire list (as one object).

Exercise 5: Program a for loop

Do Exercise 3, but use a for loop.

Exercise 6: Create a list with a for loop

Do Exercise 4, but use a for loop.

Exercise 7: Write a Python function

Write a function `x(n)` for computing an element in the sequence $x_n = n^2 + 1$. Call the function for $n = 4$ and write out the result.

Exercise 8: Return three values from a Python function

Write a Python function that evaluates the mathematical function $f(x) = \cos x$, $f'(x) = -\sin x$, and $f''(x) = -\cos x$. Return these three values. Write out the results of these values for $x = \pi$.

Exercise 9: Plot a function

Make a program that plots the function $g(y) = e^{-y}\sin(4y)$ for $y \in [0, 4]$. Use 501 points along the curve and store these points in arrays.

Exercise 10: Measure the efficiency of vectorization

IPython[6] an enhanced interactive shell for doing computing with Python. IPython has some user-friendly functionality for quick testing of the efficiency of different Python constructions. Start IPython by writing `ipython` in a terminal window. The interactive session below demonstrates how we can use the timer feature `%timeit` to measure the CPU time required by computing $\sin(x)$, where x is an array of 1M elements, using scalar computing with a loop (function `sin_func`) and vectorized computing using the `sin` function from `numpy`.

```
In [1]: import numpy as np

In [2]: n = 1000000

In [3]: x = np.linspace(0, 1, n+1)

In [4]: def sin_func(x):
   ...:     r = np.zeros_like(x)   # result
   ...:     for i in range(len(x)):
   ...:         r[i] = np.sin(x[i])
   ...:     return r
   ...:

In [5]: %timeit y = sin_func(x)
1 loops, best of 3: 2.68 s per loop

In [6]: %timeit y = np.sin(x)
10 loops, best of 3: 40.1 ms per loop
```

Here, `%timeit` ran our function once, but the vectorized function 10 times. The most relevant CPU times measured are listed, and we realize that the vectorized code is $2.68/(40.1/1000) \approx 67$ times faster than the loop-based scalar code.

Use the recipe above to investigate the speed up of the vectorized computation of the $s(t)$ function in Section 2.1.

---

[6]http://ipython.org/