Introduction to Python

as part of the course
Introduction to Complex Systems (BETA B1CS 2025-2026)

Reyk Börner (r.borner@uu.nl)
based on the 2024 slides from René van Westen and Robbin Bastiaansen

Last edited: 9 February 2026

Python and Jupyter notebooks

Python is a programming language, unless it is a member of the *Pythonidae* family of nonvenomous snakes.

There are other languages like C, Julia, Fortran, MatLab, Mathematica. Python has an intuitive syntax, is free and open-source, and has a wide range of packages that make it very powerful.

> You can download Python in different ways. We recommend installing the Anaconda Distribution (https://docs.anaconda.com/getting-started/), which includes useful software (including Jupyter notebook, an application that allows you to make notebooks like this one) and automatically pre-installs the main packages we'll need (numpy, matplotlib).

Python scripts have the file ending `.py` .

Jupyter notebooks

This document was written as a Jupyter notebook. Jupyter notebooks allow you to blend code with formatted text, which can be very useful for documenting and communicating your code (even for handing in programming exercises!).

> For more infos on installing and using Jupyter, see here.

Jupyter notebooks have the file ending `.ipynb` . In a notebook, computations can be run as code snippets in individual cells, each giving individual output. To run a cell, press `shift+enter` .

Besides code cells, you can also add Markdown cells like this one (double click to see the syntax), which you can use to structure and explain your code. You can even write math (using LaTeX notation) to tell math jokes like $\varepsilon < 0$ or formulate ground-breaking equations like

$$a^2 + b^2 = c^2 \, .$$

Now, back to Python!

Basic syntax

Displaying output

To explicitly show output, you can use the `print()` function.

To suppress output (in Jupyter notebooks, the last command evaluated in a cell is printed below the cell), you can put a `;` at the end of the line of code.

Commenting code

Code readability is crucial! Comments help you and others understand what the code does.
The comment environment is stared with a `#` symbol. Anything in the line following `#` will not be evaluated by Python.

For a longer comment across multiple lines, you can also use the environment `""" bla bla bla """`.

In [1]:
```
# This is a comment, Python will not evaluate this.

"""
This is a longer comment.
""";
```

Types: Strings, integers, floats

A string stores text. It is defined by enclosing it in (single or double) quotation marks.

In [2]:
```
'This is a string'
"This is also a string";
```

We can check the type of something using the built-in `type()` function.

In [3]:
```
type('This is a string')
```

Out[3]:
```
str
```

Integers and real numbers (*floating point* numbers) have different types.

In [4]:
```
type(8) # Integer
```

Out[4]:
```
int
```

In [5]:
```
type(8.0) # Float
```

Out[5]:
```
float
```

Defining variables

It is good practice to store strings and numbers as variables with a given name. You can then refer to the variable via its name.

In [6]:
```
# Define a string named my_string
my_string = 'Life is complex.'

print(my_string)
```

```
Life is complex.
```

In [7]:
```
# Define the number pi (up to the sixth decimal)
pi = 3.141592

print(pi)
```

```
3.141592
```

In [8]:
```
# Check what type pi is
```

```
print(type(pi))
```

```
<class 'float'>
```

You can overwrite any variable by re-defining it.

Basic arithmetic operations

Addition, subtraction, multiplication, division, exponentiation - you can use these operations on integers and floats.
Addition and multiplication also works for strings, but with a different meaning.

In [9]:
```
# Add
2+3
```

Out[9]:
```
5
```

In [10]:
```
# Subtract
2-5
```

Out[10]:
```
-3
```

In [11]:
```
# Multiply
2*6
```

Out[11]:
```
12
```

In [12]:
```
# Divide
3/2
```

Out[12]:
```
1.5
```

In [13]:
```
# Exponentiate
# NOTE: The syntax for "a to the power of b" is a**b and not a^b as you might expect.
2**3
```

Out[13]:
```
8
```

Adding strings concatenates them. Multiplying a string with an integer concatenates the string multiple times.

In [14]:
```
my_string + my_string
```

Out[14]:
```
'Life is complex.Life is complex.'
```

In [15]:
```
my_string*4
```

Out[15]:
```
'Life is complex.Life is complex.Life is complex.Life is complex.'
```

Conditional statements: If and else

If statements allow you to write code that does different things depending on a given condition.

For example, let us write a program that checks whether the number `pi` defined above is an integer.

In [16]:
```
if type(pi) == int:
    print('Pi is an integer')
else:
    print('Pi is not an integer')
```

```
Pi is not an integer
```

Note the syntax: To test the equality of two things, we use `==` . At the end of the `if` and `else` lines, there must be a `:` .

Comparisons:

- `==` equals
- `!=` does not equal
- `>` is greater than
- `<` is less than
- `>=` is greater than or equal
- `<=` is less than or equal

If, elif and else statements

Sometimes you need more than two options. For this, you can use `elif` ("or else, if...")

In [17]:
```python
if type(pi) == int:
    print('Pi is an integer')
elif type(pi) == float:
    print('Pi is a float')
else:
    print('Pi is neither an integer nor a float')
```

```
Pi is a float
```

You can also nest if statements (one inside the other). For example, let us write code that checks whether a variable is a floating point number and if so, whether it is positive, negative or zero.

In [18]:
```python
number = -1.2

if type(number) != float:
    print('Not a floating point number.')
else:
    if number > 0:
        print('The number is positive')
    elif number < 0:
        print('The number is negative')
    else:
        print('The number is zero')
```

```
The number is negative
```

Loops

While loops

We will often need to repeat certain blocks of code several times. One way to achieve this is using a `while` loop.

A `while` loop runs the code inside the loop over and over again as long as a given **condition** is fulfilled. For example, let us print all natural numbers up to 7.

In [19]:
```python
number = 0
while number < 8:            # define the condition
    # start
    print(number)            # print the number
    number = number + 1      # increase the number by one
    # jump back to start
    # and check condition
```

```
0
1
2
3
4
5
6
7
```

As you see, the loop stops once the condition is **not** fulfilled.

> WARNING: As long as the condition in the `while` statement is fulfilled, it will keep looping forever (and crash Python). For example, if you forget increasing `number` by including the line `number = number + 1`, this loop will never stop.
>
> If you have a supercomputer in the cellar, `while True:` is one way to heat the house.

For loops

Another way to run a code block repeatedly is by using a `for` loop. A `for` loop iterates through a collection of items. To loop through an interval of numbers, you can use the `range()` function.

The following achieves the same as the previous `while` loop:

In [20]:
```
for number in range(0,8):
    print(number)
```

```
0
1
2
3
4
5
6
7
```

> Note: In Python, we start counting at `0`, and the end of the range is excluded, so `range(0,8)` goes from 0 to 7.

Nested for loops

Just like `if` statements and `while` loops, you can also nest `for` loops.

In [21]:
```
for i in range(3):
    for j in range(3):
        print(i, j)
```

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

Here we simply wrote `range(3)` instead of `range(0,3)`, which yields the same.

Lists

Often we want to collect several elements in one variable. We can do this with lists. Lists can contain elements of different types (though that might make it difficult to work with).

> For numerical data, arrays are much better suited, as we'll learn further below.

In [22]:
```
my_list = ['hello', 8, 9.0]
print(my_list)
```

```
['hello', 8, 9.0]
```

Indexing and slicing

You can access the first item in the list by *indexing*.

In [23]:
```
print(my_list[0])   # First item in my_list
print(my_list[1])   # Second item
print(my_list[-1])  # Last item
```

```
hello
8
9.0
```

With *slicing*, you can select a certain range from a list.

In [24]:
```
print(my_list[0:2]) # Print the first two items
print(my_list[-2:]) # Print the last two items
```

```
['hello', 8]
[8, 9.0]
```

You can also loop through the items using a `for` loop:

In [25]:
```
for item in my_list:
    print(item)
```

```
hello
8
9.0
```

List operations

You can add a new item to the end of a list using the `append()` function. This updates the list under the same name.

In [26]:
```
# Add my_string to the list
my_list.append(my_string)

print(my_list)
```

```
['hello', 8, 9.0, 'Life is complex.']
```

Adding two lists simply concatenates them.

In [27]:
```
list1 = [0,1,2,3]
list2 = [5,6,7,8]

list1+list2
```

Out[27]:
```
[0, 1, 2, 3, 5, 6, 7, 8]
```

Packages

Much of the functionality that makes Python powerful is not built-in, but needs to be loaded as extra packages. There are many packages for different purposes (you can even create your own!).

Importing packages

A package can be imported using the `import` command. You can give the package a custom name via the syntax `import <package> as <your_name>`.

Numpy

For numerical programming, one of the most important packages is `numpy`.

In [28]:
```
# Import numpy
# and refer to it as "np"
import numpy as np
```

This way, you only need to type `np` and not `numpy` each time you call a function from the `numpy` package.

Numpy arrays

As discussed above, lists allow you to collect different elements in one object. They allow combining different types in one list, but are not well suited for mathematical operations with the elements of the list. Often, we want to do *element-wise* operations with a collection of elements. Numpy *arrays* are perfect for this: they come with a lot of functionality and handling large numpy arrays is much more efficient than handling large lists.

In [29]:
```
# Convert a list into a numpy array
array1 = np.array(list1)

print(array1)
print(type(array1))
```

```
[0 1 2 3]
<class 'numpy.ndarray'>
```

Indexing and slicing works in the same way as for lists.

Array operations

However, unlike with lists, we can now do *element-wise* mathematical operations with numpy arrays:

In [30]:
```
# multiply each element by 2
doubled_array = 2*array1
print(doubled_array)
```

```
[0 2 4 6]
```

In [31]:
```
# Exponentiate each element by 3
cubed_array = array1**3
print(cubed_array)
```

```
[ 0  1  8 27]
```

In [32]:
```
array2 = np.array(list2)
print(array1)
print(array2)

# Add two arrays elementwise (must be of same length)
print(array1 + array2)
```

```
[0 1 2 3]
[5 6 7 8]
[ 5  7  9 11]
```

Note that adding two arrays is very different from adding two lists!

In [33]:
```
# Elementwise exponentiation (array2 to the power of array1)
print(array2**array1)
```

```
[  1   6  49 512]
```

Creating arrays

Often you'll need to create an array from scratch.

In [34]:
```
# Array of 10 zeros
array = np.zeros(10)
print(array)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

In [35]:
```
# Array of 10 ones
array = np.ones(10)
print(array)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

In [36]:
```
# Array of integers between 10 and 19
array = np.arange(10,20)
print(array)
```

```
[10 11 12 13 14 15 16 17 18 19]
```

In [37]:
```
# Array of 11 equally spaced points between 0 and 1
array = np.linspace(0, 1, 11)
print(array)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

To get the length of an array, use the `len()` function or the `shape()` function.

In [38]:
```
print(len(array))
print(array.shape[0]) # The index 0 returns the length along the first dimension. Here
```

```
11
11
```

> Read more in the full Numpy documentation.

Example: Interest on your savings

Suppose you have `x` Euros. Your high-achieving friend Myrthe has `y` Euros. Each of you deposit their money in a sketchy bank that offers 3% interest per day. How do your savings grow over the first 100 days?

In [39]:
```
N_days = 100  # number of days
rate = 1.03   # Interest rate
x = 2         # Your start capital (Euros)
y = 5         # Myrthe's start capital (Euros)

# Set up arrays to store the data
my_savings = np.ones(N_days)*x
myrthe_savings = np.ones(N_days)*y

# Calculate the account balance at each day
# based on the previous day
for day in range(1,N_days):
    my_savings[day] = my_savings[day-1]*rate
    myrthe_savings[day] = myrthe_savings[day-1]*rate

print(my_savings)
```

```
[ 2.          2.06        2.1218      2.185454    2.25101762  2.31854815
  2.38810459  2.45974773  2.53354016  2.60954637  2.68783276  2.76846774
  2.85152177  2.93706743  3.02517945  3.11593483  3.20941288  3.30569526
  3.40486612  3.50701211  3.61222247  3.72058914  3.83220682  3.94717302
  4.06558821  4.18755586  4.31318254  4.44257801  4.57585535  4.71313101
  4.85452494  5.00016069  5.15016551  5.30467048  5.46381059  5.62772491
  5.79655666  5.97045336  6.14956696  6.33405397  6.52407558  6.71979785
  6.92139179  7.12903354  7.34290455  7.56319168  7.79008743  8.02379006
  8.26450376  8.51243887  8.76781204  9.0308464   9.30177179  9.58082494
  9.86824969 10.16429718 10.4692261  10.78330288 11.10680197 11.44000603
 11.78320621 12.13670239 12.50080347 12.87582757 13.2621024  13.65996547
 14.06976443 14.49185737 14.92661309 15.37441148 15.83564382 16.31071314
 16.80003453 17.30403557 17.82315664 18.35785134 18.90858688 19.47584448
 20.06011982 20.66192341 21.28178111 21.92023455 22.57784158 23.25517683
 23.95283214 24.6714171  25.41155961 26.1739064  26.95912359 27.7678973
 28.60093422 29.45896225 30.34273111 31.25301305 32.19060344 33.15632154
 34.15101119 35.17554152 36.23080777 37.317732  ]
```

We can read off our savings above, but it helps to visualize the data.

Plotting

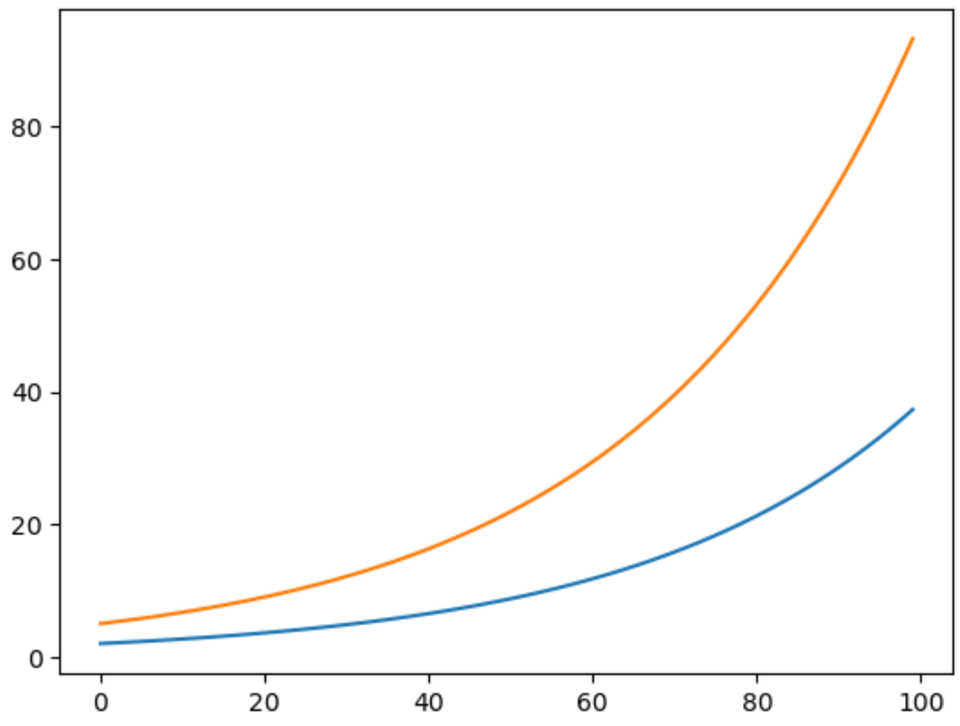
For this, we need to load another package that provides graphical visualization functionality. The most popular one in Python is `pyplot`, a subpackage of `matplotlib`.

In [40]:
```
# Load pyplot
import matplotlib.pyplot as plt
```

Example: Savings

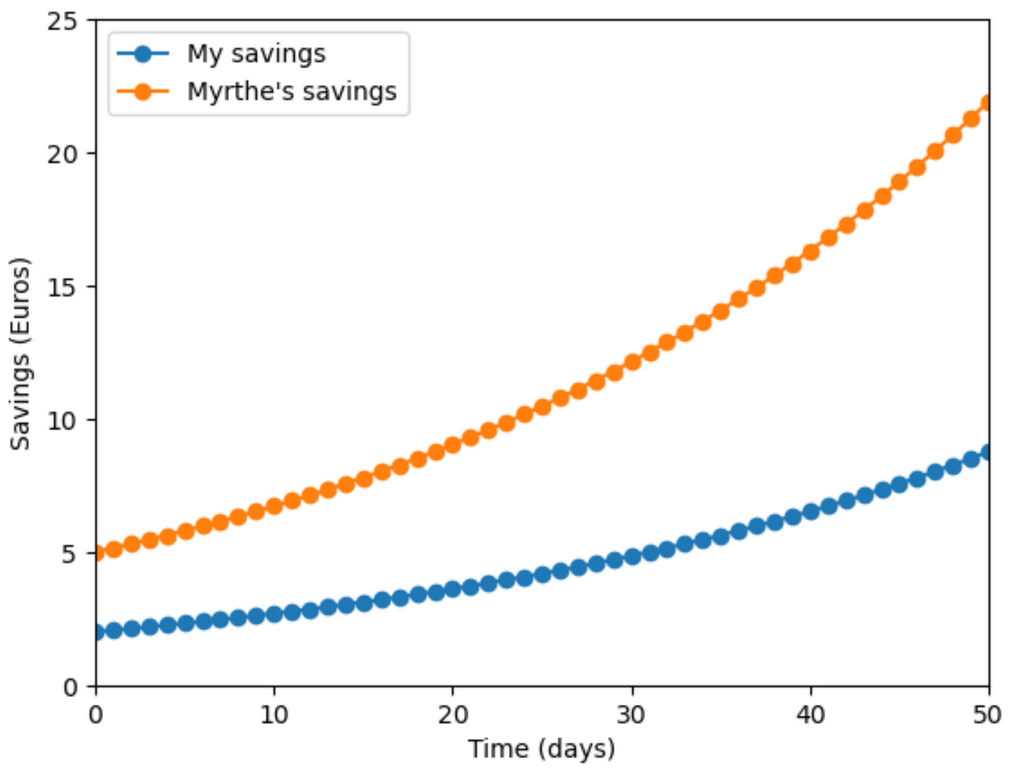
The code below exemplifies how we can plot your and Myrthe's savings over time.

In [41]:
```
plt.figure()                                      # Create new figure
plt.plot(np.arange(N_days), my_savings)           # Add a line plot of my savings over tim
plt.plot(np.arange(N_days), myrthe_savings)
plt.show()                                        # Display the figure
```



This shows that the savings grow exponentially, but without context, nobody would understand this plot. We need to add axis labels and a legend. Also, suppose we only want to plot the savings over the first 50 days.

```
In [42]: plt.figure()
         plt.plot(np.arange(N_days), my_savings, "-o", label="My savings")
         plt.plot(np.arange(N_days), myrthe_savings, "-o", label="Myrthe's savings")
         plt.xlabel("Time (days)")
         plt.ylabel("Savings (Euros)")
         plt.xlim(0,50)
         plt.ylim(0,25)
         plt.legend()
         plt.show()
```



Here we have used

- `xlabel()` and `ylabel()` to define the axis labels
- `xlim()` and `ylim()` to define the limits of the x and y axis, respectively
- the shortcut argument `"-o"` in the `plot()` function to show each data point with lines connecting them
- the `label=` keyword argument to specify the label that will show in the legend
- the `legend()` function to display the legend

> Check out the extensive Matplotlib documentation for more!

Functions

Often we will want to use a certain block of code several times, possibly in different ways. For this purpose, functions are very powerful. They also make your code much cleaner, easier to read, and less prone to errors (because you avoid duplicate code).

In fact, all the methods we have used so far from the `numpy` and `matplotlib.pyplot` packages, such as `np.arange(...)` or `plt.plot(...)` are functions themselves.

The general syntax for defining functions is:

```
def function_name(arg1, arg2, ..., kwarg1=default1, kwarg2=default2, ...):
    # do something
    return output
```

Here `arg1` and `arg2` are *positional* arguments. Their order inside `function_name()` matters and they are mandatory arguments. `kwarg1` and `kwarg2` are *keyword* arguments: They are labelled by a keyword (such as `kwarg1`) and are optional. If they are not specified by the user when calling the function, the default value (here `default1`) is used.

The simplest function is a function without any input argument, which can only do one particular thing. For example, let us write a function that answers a simple question about life.

In [43]:
```
# Define the function
def what_is_life():
    print("It's complex.")

# Call the function
what_is_life()
```

```
It's complex.
```

Now you know!

Positional arguments

Suppose we want to plot a parabola, $y = f(x)$ with $f(x) = x^2$. For this, we need a function that squares whatever number `x` you input as an argument.

In [44]:
```
# Defining the function
def parabola(x):
    y = x**2
    return y

# Calling the function for different values
print(parabola(3))
print(parabola(-4))
```

```
9
16
```

So this is a function with one positional argument, `x`. Calling the function manually for each x value is of course not what we want. Luckily, a function like `parabola` also works on `numpy` arrays by calling the function element-wise.

```
In [45]: # Create array of x values
x_values = np.arange(-5,6)

# Calculate y values
y_values = parabola(x_values)

print("x_values: ", x_values)
print("y_values: ", y_values)
```

```
x_values:  [-5 -4 -3 -2 -1  0  1  2  3  4  5]
y_values:  [25 16  9  4  1  0  1  4  9 16 25]
```

The function $f(x) = x^2$ is a special parabola with curvature 1 and a minimum at 0. The general form for a parabola is $f(x) = ax^2 + b$, where a is the curvature and b the y offset.

Positional arguments

We can use keyword arguments to write a function that, by default, returns a standard parabola of the form $f(x) = x^2$, while the curvature and offset can be optionally adjusted:

```
In [46]: # Define the function
def parabola2(x, a=1, b=0):
    """
    Returns a*x^2 + b

    Keyword arguments
    - a=1: Curvature
    - b=0: y offset
    """

    y = a*x**2 + b
    return y
```

Docstrings

In the function definition, we have used the comment environment `"""..."""` to write a *docstring* for the function. Docstrings document what the function does and what its inputs are.

You can get the docstring of a function by typing `? function_name()`, which is especially handy for quickly looking at the documentation of functions from external packages.

```
In [47]: ? parabola2
```

```
Signature:  parabola2(x, a=1, b=0)
Docstring:
Returns a*x^2 + b

Keyword arguments
- a=1: Curvature
- b=0: y offset
File:      /var/folders/rn/ycx08v950x3dl8vm_j9jvz200000gn/T/ipykernel_18048/3689547486.p
y
Type:      function
```

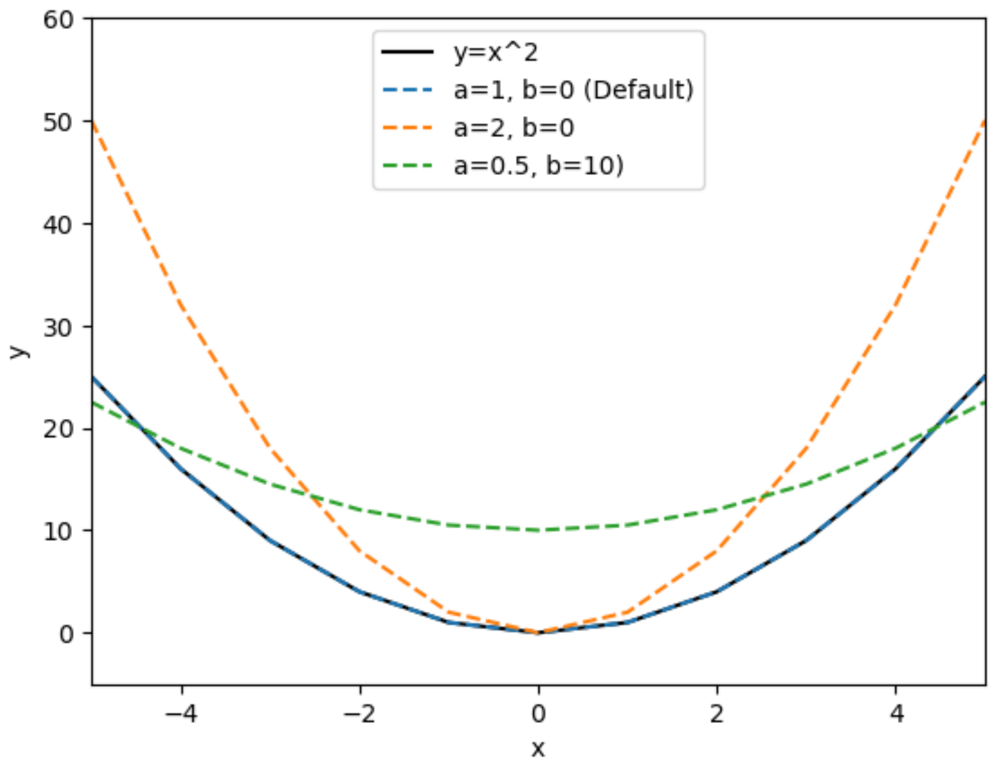
```
In [48]: ? round
```

```
Signature:  round(number, ndigits=None)
Docstring:
Round a number to a given precision in decimal digits.

The return value is an integer if ndigits is omitted or None.  Otherwise
the return value has the same type as the number.  ndigits may be negative.
Type:      builtin_function_or_method
```

Plotting (continued)

Let's make some plots to see how the function `parabola2` behaves. Remember, to plot something you must first load the pyplot package (`import matplotlib.pyplot as plt`). We have already done this above.

In [49]:
```
# Call function with different input arguments
y_values2 = parabola2(x_values)
y_values3 = parabola2(x_values, a=2)
y_values4 = parabola2(x_values, a=0.5, b=10)

# Plot results
plt.plot(x_values, y_values, label="y=x^2", c="black")
plt.plot(x_values, y_values2, linestyle="dashed", label="a=1, b=0 (Default)")
plt.plot(x_values, y_values3, linestyle="dashed", label="a=2, b=0")
plt.plot(x_values, y_values4, linestyle="dashed", label="a=0.5, b=10)")
plt.xlabel("x")
plt.ylabel("y")
plt.xlim(-5,5)
plt.ylim(-5,60)
plt.legend()
plt.show()
```



Note that here we have specified the style and color of some of the lines using the `c` (color) and `linestyle` (short form `ls`) keyword arguments of the `plt.plot` function.

As your code becomes more complex, functions will become increasingly important to make the code flexible and structured. The choice of positional arguments and keyword arguments is a question of code design. Generally, it proves practical to use positional arguments for main parameters that users are expected to always specify when using the function. Keyword arguments are great for options that may only be used sometimes or have a common default value. Also, if you have more than 3-4 positional arguments, it can become confusing in which order the arguments must be specified. Keyword arguments might be more appropriate then, since they explicitly label each argument.

Working with `numpy` and `matplotlib`

Random numbers

Sometimes you might need to generate random numbers, for example to simulate fluctuations (noise) in a system or simply to test some code. The `numpy` package includes useful functionality for random number generation.

For example, we can sample random numbers from the interval `[0,1]` with uniform probability distribution using `np.random.rand(N)`, where `N` is the number of (independent) random numbers generated.

In [50]:
```
# Uniformly distributed
uniform = np.random.rand(3)
print(uniform)
```

```
[0.01756357 0.19867911 0.28334763]
```

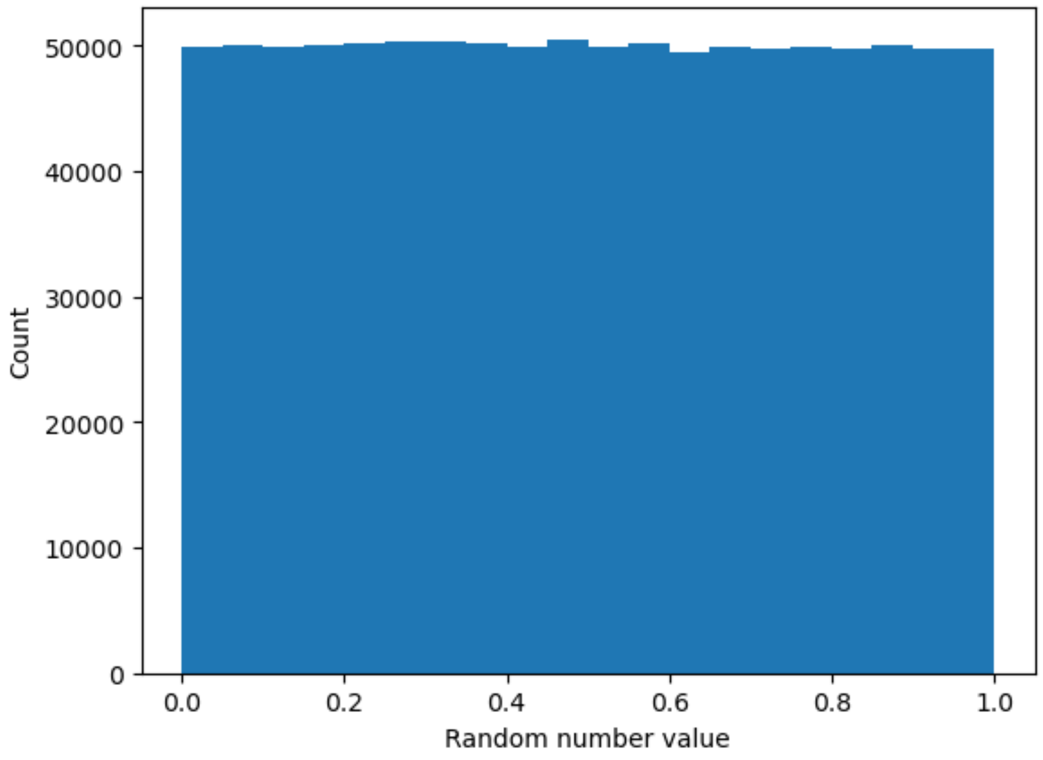
If you need to double check what the `rand` function of `np.random` does, look at the docstring:

In [51]:
```
? np.random.rand
```

```
Signature:  np.random.rand(*args)
Docstring:
rand(d0, d1, ..., dn)

Random values in a given shape.

.. note::
    This is a convenience function for users porting code from Matlab,
    and wraps `random_sample`. That function takes a
    tuple to specify the size of the output, which is consistent with
    other NumPy functions like `numpy.zeros` and `numpy.ones`.

Create an array of the given shape and populate it with
random samples from a uniform distribution
over ``[0, 1)``.

Parameters
----------
d0, d1, ..., dn : int, optional
    The dimensions of the returned array, must be non-negative.
    If no argument is given a single Python float is returned.

Returns
-------
out : ndarray, shape ``(d0, d1, ..., dn)``
    Random values.

See Also
--------
random

Examples
--------
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
Type:      method
```

Are these numbers really drawn from a uniform distribiution? We can check! To do this, let's generate 1 million random numbers, and plot them as a histogram (occurrence frequency as a function of the number value). Plotting a histogram can be achieved very simply by calling the `plt.hist` function of `pyplot`.

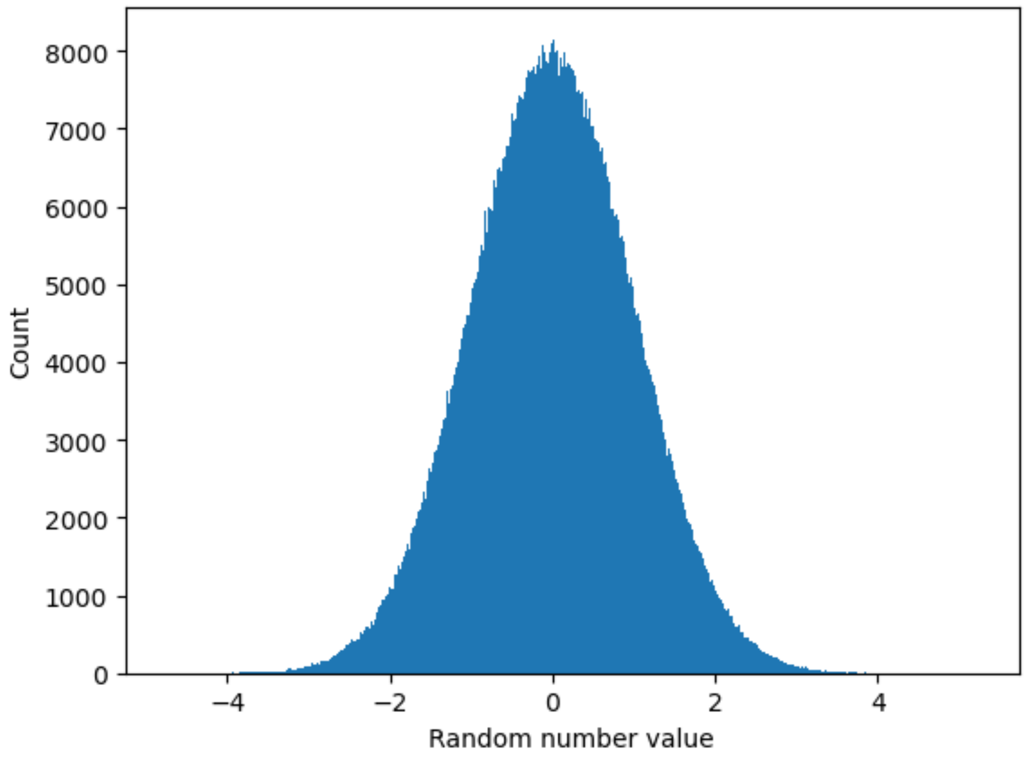
In [52]:
```
# Sample 1 million random numbers
sample = np.random.rand(int(1e6)) # 1e6 = 1 million in scientific notation (10^6)

# Plot their distribution as a histogram with 20 bins
plt.hist(sample, bins=20)
plt.xlabel("Random number value")
plt.ylabel("Count");
```



As expected, the distribution is almost constant as a function of the number value. The slight deviations are because we have a finite sample size of 1 million. The more samples we generate, the more uniform the distribution will become (try it!).

Similarly, we can generate normally distributed numbers (sampled from a Gaussian distribution with mean 0 and standard deviation 1):

In [53]:
```
# Normally distributed
sample_normal = np.random.randn(int(1e6))

plt.hist(sample_normal, bins=500)
plt.xlabel("Random number value")
plt.ylabel("Count");
```



We get a nice Gaussian bell curve, as expected.

When working with random numbers, one issue is that the results are not automatically reproducible, because the random numbers will change each time you run the code again. To avoid this, you can specify a *random number seed*, which always returns the same sequence of random numbers.

In [54]:
```
# Set random number seed
np.random.seed(1234) # Here 1234 is the seed ID

random_number = np.random.rand()
print(random_number)
```

```
0.1915194503788923
```

Now, each time you run the above cell of code, you will get exactly `0.1915...` , unless you change the seed ID.

2D arrays

So far, we have introduced `numpy` arrays as *vectors* that allow efficient mathematical operations on them. These arrays were one-dimensional. We can also create higher-dimensional arrays. For example, a 2D numpy array corresponds to a *matrix*.

In [55]:
```
# 1D array
array_1d = np.arange(5) # A vector with 5 elements
print(array_1d)
print("Shape: ", array_1d.shape)

# 2D array
array_2d = np.ones((3,5)) # A matrix of ones with 3 rows and 5 columns
print(array_2d)
print("Shape: ", array_2d.shape)
```

```
[0 1 2 3 4]
Shape:  (5,)
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
Shape:  (3, 5)
```

Note that the shape of `array_2d` is specified as a *tuple* `(3,5)`.

Let's say we want to overwrite the first row of `array_2d` with a different vector.

In [56]:
```
# New vector (1D array)
vector = np.arange(5)

# Overwrite first row with the vector
array_2d[0,:] = vector

print(array_2d)
```

```
[[0. 1. 2. 3. 4.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

We can access an element of the vector by indexing, or multiple elements by slicing:

In [57]:
```
# First row, fourth column
print(array_2d[0,3])

# First three entries of first row
print(array_2d[0,:3])
```

```
3.0
[0. 1. 2.]
```

Numpy provides a lot of functionality for arrays of 1, 2 and more dimensions. For example, suppose we want to compute the average of all entries in the array.

In [58]:
```
# Take the mean (average) over the whole array
mean = np.mean(array_2d)

print("Mean: ", mean)
```

```
Mean:  1.3333333333333333
```

Often, we only want to compute the mean along a certain dimension. Say, the `array_2d` gives temperature measurements, where each row corresponds to a different location in the Netherlands and each column corresponds to a month, from January to May. We might then ask about the average temperature at a given location over time, or about the spatial average temperature during a given month.

In [59]:
```
# Average over time (along rows)
mean_time = np.mean(array_2d, axis=1)

# Average over locations (along columns)
mean_loc = np.mean(array_2d, axis=0)

print("Temporal mean at each location: ", mean_time)
print("Spatial mean during each month: ", mean_loc)
```

```
Temporal mean at each location:  [2. 1. 1.]
Spatial mean during each month:  [0.66666667 1.         1.33333333 1.66666667 2.
]
```

> For more functionality, check out the numpy documentation.

> If you are actually working with measurement data like in the example above, it is very useful to work with *named arrays*, i.e. arrays in which the rows, columns and entries can be labelled - and accessed using these labels (like "time" and "location" rather than abstract indices 0 or 1). This goes beyond our tutorial, but for reference, great packages offering such functionality are `pandas` and `xarray`.

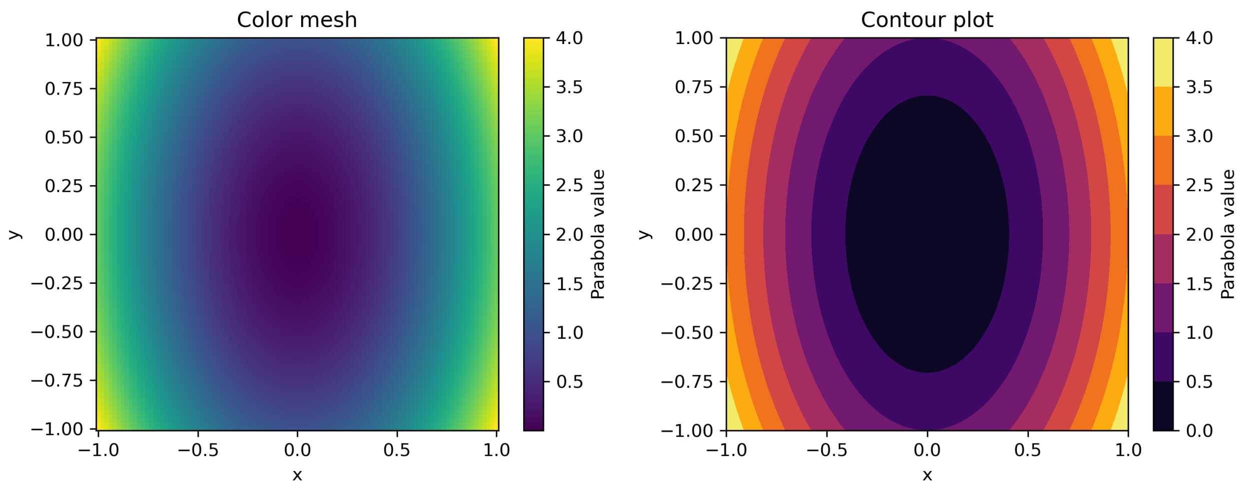
Plotting 2D fields

For two-dimensional data, simple line plots like the ones above are often not the most suitable choice. For example, suppose we want to plot a parabolic surface as a function of x and y. We can achieve this in multiple ways, as shown below. We will also take this opportunity to exemplify how we can create more complex plots.

In [60]:
```
# x and y values
x = np.linspace(-1,1,100) # 100 equally spaced numbers from -1 to 1
y = np.linspace(-1,1,100)

# Create a 2D grid of x and y values
xx, yy = np.meshgrid(x, y)

# define the parabolic surface
def parabola_2D(x, y, a=3):
    return a*x**2 + y**2
```

In `pyplot`, figures are composed of a `figure()` object (the "canvas"), which contains one or more `axis` objects (a plot panel). These objects can be defined together using the `subplots()` function.

In [61]:
```
# Create figure and 1 row with 2 axes
fig, ax = plt.subplots(1,2, figsize=(10,4), dpi=300) # dpi = figure resolution (dots pe

# Left panel: Color mesh
left = ax[0].pcolormesh(xx, yy, parabola_2D(xx, yy))
fig.colorbar(left, ax=ax[0], label="Parabola value")
ax[0].set_title("Color mesh")

# Left panel: Contour plot
right = ax[1].contourf(xx, yy, parabola_2D(xx, yy), cmap="inferno") # cmap specifies th
fig.colorbar(right, ax=ax[1], label="Parabola value")
ax[1].set_title("Contour plot")

for axs in ax: # Loop through all axes
    axs.set(xlabel="x", ylabel="y")

fig.tight_layout() # Automatically adjusts positioning of subpanels and labels
plt.show()
```

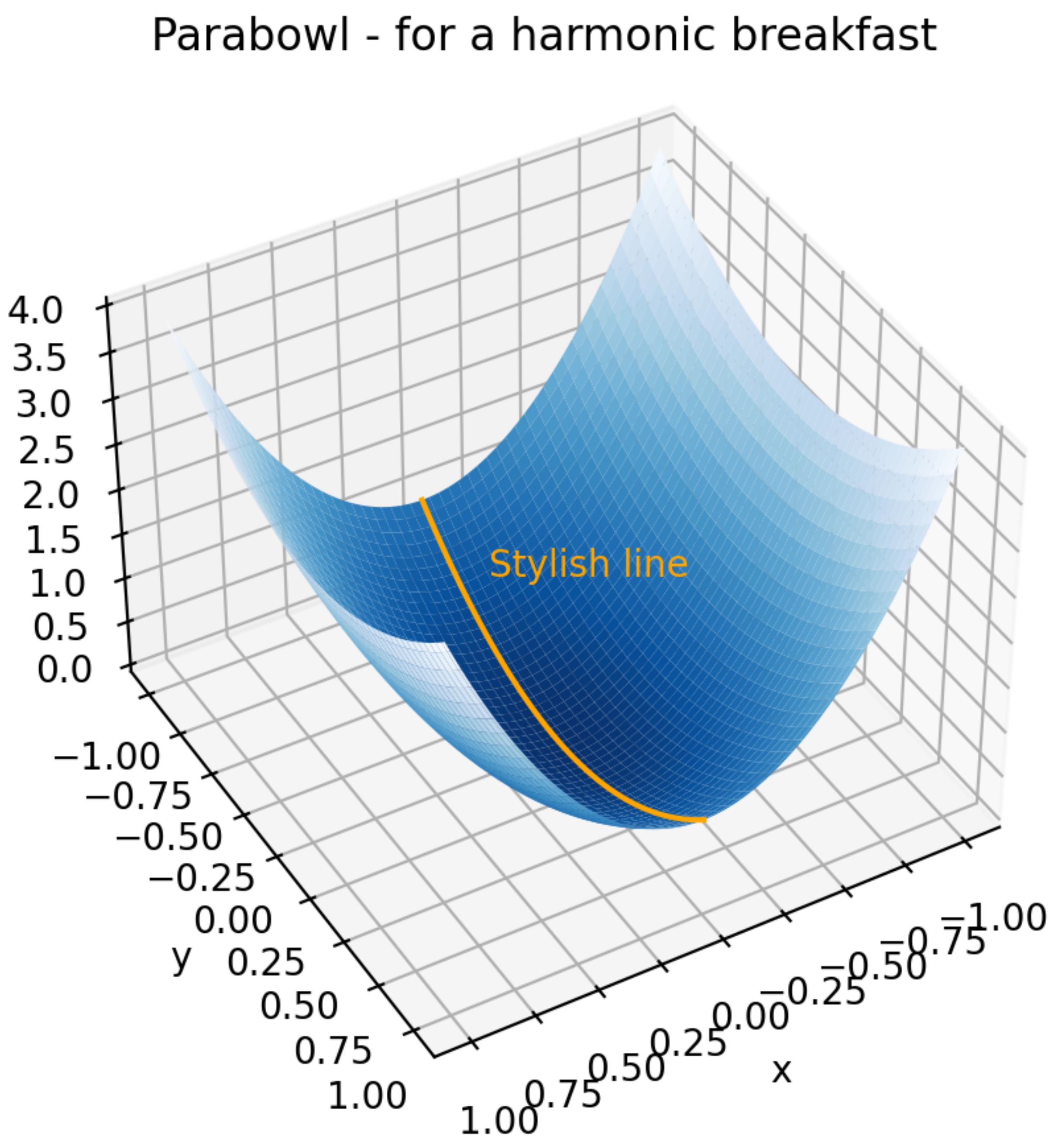


The left and right panel are plotting the same data, but in different ways: on the left, we have a continuous color shading using the default colormap `viridis` . On the right, the color shading is discrete, showing filled contours using the `inferno` colormap.

> A note on colormaps: The color representation of your data can make a big difference for how people interpret the data. There are many colormaps out there, some of which can badly distort the information, so be mindful of your choice! Both colormaps plotted above are *perceptually uniform*, colorblind-friendly and still make sense when printed in black and white.

Suppose the 2D parabola describes your latest design for a cereal bowl (not the most functional choice, but hey). To convince your customers of the design, the 2D plots above are not ideal. Wouldn't it be much better to present a 3D model? With matplotlib, that's not a problem!

In [62]:
```
# Create a 3D plot
fig = plt.figure(figsize=(5,5), dpi=200)
ax = fig.add_subplot(projection="3d") # creates a 3D axis
ax.plot_surface(xx, yy, parabola_2D(xx, yy), cmap="Blues_r")
ax.plot(np.zeros(100), y, parabola_2D(0,y), c="orange", zorder=3) # zorder determines w
ax.text(0, -0.5, 1, "Stylish line", color="orange") # this adds a text annotation at po
ax.set(xlabel="x", ylabel="y", title="Parabowl - for a harmonic breakfast")
ax.view_init(40,60) # viewing angle
plt.show()
```



Coding tips: Best practices

Good code is...

- Clear
- Easy to understand
- Documented
- Reproducible
- Reliable
- Reusable
- Extendable

1. Document your code

Use short comments `# My comment` and long comments `"""..."""` to document and explain your code. Do not overuse comments though. For example,

```
1 + 2 # add 1 and 2
```
is superfluous. Generally, the code should speak for itself; comments help to describe the purpose of the code on a higher level, e.g.

```
1 + 2 # Total number of apples
```
Each function can be documented with a *docstring*, as already mentioned above:

In [63]:
```
def my_function(x, a=1):
    """
    Computes the square of x, times a prefacor.

    ## Keyword arguments
    - a = 1: The prefactor
    """
    return a*x**2
```

In [64]:
```
# Display docstring
? my_function
```

```
Signature:  my_function(x, a=1)
Docstring:
Computes the square of x, times a prefacor.

## Keyword arguments
- a = 1: The prefactor
File:      /var/folders/rn/ycx08v950x3dl8vm_j9jvz200000gn/T/ipykernel_18048/3634753278.p
y
Type:      function
```

2. Write functions

Functions help to

- organize code in readable blocks
- avoid duplicate code (writing very similar lines of code multiple times)
- make the code adaptable and extendible

Shorter code is less prone to errors, so try to keep your code as concise as possible! Functions are great for this purpose.

You can also split up a larger task into multiple functions and then write a main function that consecutively calls each of these functions in the appropriate order. As a rough rule of thumb, if your function becomes longer than 50 lines of code, it might make sense to break it up into multiple functions.

3. Use descriptive names

The moment you need to change something in your code a few weeks after writing it is often the moment when you wish you had named your variables and functions a bit more logically.

Bad names:

- `imoutofstroopwafelsxxx`
- `fft`
- `l63a`
- `vector3`

Better names

- `curvature`
- `fast_fourier_transform`
- `lorenz63_attractor`
- `savings`

4. Write efficient code

- Avoid duplicate code. Use functions and loops so you don't have to repeat code.
- Use the functionality of `numpy` to perform element-wise operations rather than writing lots of for loops. This is much faster and less prone to errors.
- More generally, it is great to use built-in functions (as long as you know what they are doing). They are often optimized for speed and there is no need to re-invent the wheel.

Useful packages

- Popular Python packages are well-documented. Reading the documentation can save a lot of time guessing!
- Packages that I found useful (besides `numpy and matplotlib` :
  - `scipy` : optimization, fitting, machine learning, and more
  - `pandas` : data analysis
  - `xarray` : handling large datasets
  - `cartopy` : geographic plotting on maps

That's it!

There's a lot more to learn about Python, but hopefully this got you started.

> Found a mistake? Let me know!