PHOT 110: Introduction to programming

LECTURE 04

Michaël Barbier, Spring semester (2023-2024)

SUMMARY OF FOR AND WHILE

`while` loop:

- repeat under condition
- we don’t know how many iterations we need, and
- we have a stopping criterium

```
while <condition>:
        <statement>
        <statement>
        ...
        <statement>
```

`for` loop:

- repeat a process
- number of **iterations** is known, or
- we iterate over a list of elements

```
for <el> in <list>:
        <statement>
        <statement>
        ...
        <statement>
```

THE FOR LOOP SO FAR …

Iterate over a **sequence**: `list`, `range`, etc.

```python
for el in ["banana", "orange", "apple"]:
  print(el)
```

```
banana
orange
apple
```

```python
for el in range(3):
  print(el)
```

```
0
1
2
```

But . . . We can actually iterate over other types: `string`, `set`, etc.

OUTLOOK: FOR LOOPS, SEQUENCES, AND ARRAYS

- sequences: lists, ranges, tuples
- strings as special sequences
- enumerate
- list comprehension
- break, continue, and pass
- arrays
- simple plots

SEQUENCES: LISTS, RANGES, TUPLES, STRINGS

SEQUENCES

- Sequences are **collections** of elements with an **order**
- **unordered collections** exist: **sets**
- The following types are sequences:
  - **`list`**, **`range`**, strings (**`str`**)
  - **tuple** (which we will see today), …
- Common operations: indexing, slicing, `len()`, `.sort()`
- Can be mutable (adaptable) or immutable

MUTABLE VS. IMMUTABLE

- Every object has a **unique ID** (CPython: address in memory)

```
i = 45
print(id(i))
```

```
140714479177528
```

```
a_string = "Pretzel"
a_list = [3, 105, 56]
a_range = range(4)
print(f"The ID of a_string: {a_string} = { id(a_string) }
print(f"The ID of a_list: {a_list} = { id(a_list) }")
print(f"The ID of a_range: {a_range} = { id(a_range) }")
```

```
The ID of a_string: Pretzel = 2135801745536
The ID of a_list: [3, 105, 56] = 2135519091008
The ID of a_range: range(0, 4) = 2135801181456
```

MUTABLE VS. IMMUTABLE

- Numbers are immutable
- If you change them, the ID will change
- Even if you assign again the same variable name

```
i = 45
print(id(i))
i = i + 1
print(id(i))
```

```
140714479177528
140714479177560
```

MUTABLE VS. IMMUTABLE

- strings (`str`) are immutable
- Operations will provide a copy

```python
A = "Pretzel"
B = A.upper()
print(f"String A: {A} is still the same as A: {A}")
print(f"String B: {B} is the uppercase version of A: {A}"
print(f"ID of B: {id(B)} is not equal to ID of A: {id(A)}
```

```
String A: Pretzel is still the same as A: Pretzel
String B: PRETZEL is the uppercase version of A: Pretzel
ID of B: 2135801527968 is not equal to ID of A:
2135801745536
```

MUTABLE VS. IMMUTABLE

- Lists are mutable
- Elements can change, added, removed, etc.

```
a_list = [3, "car", 5]
print(a_list)
print(id(a_list))

# Change 2nd element
a_list[1] = "tree"
print(a_list)
print(id(a_list))
```

```
[3, 'car', 5]
2135519086464
[3, 'tree', 5]
2135519086464
```

TUPLES ARE IMMUTABLE LISTS

Syntax to define a tuple:

```python
t = (3, "leaf", False)
print(t)
```

```
(3, 'leaf', False)
```

Cast another collection to tuple, e.g. from `list`:

```python
t = tuple([1, 3.23, 3, 5 + 6J])
print(t)
```

```
(1, 3.23, 3, (5+6j))
```

Tuples are immutable:

```python
t = tuple([1, 3.23, 3, 5 + 6J])
t[0] = 2
```

```
TypeError: 'tuple' object does not support item assignment
```

TUPLES ARE IMMUTABLE LISTS

Lists are mutable

```python
a_list = [3, "car", 5]
print(a_list)

# Change 2nd element
a_list[1] = "tree"
print(a_list)
```

```
[3, 'car', 5]
[3, 'tree', 5]
```

Tuples are immutable

```python
a_tuple = (3, "car", 5)
print(a_tuple)

# Change 2nd element
a_tuple[1] = "tree"
print(a_tuple)
```

```
(3, 'car', 5)
```

```
TypeError: 'tuple' object
does not support item
assignment
```

TUPLES ARE IMMUTABLE LISTS

Lists are mutable

```
a_list = [3, "car", 5]
print(a_list)

# Append element
a_list.append(True)
print(a_list)
```

```
[3, 'car', 5]
[3, 'car', 5, True]
```

Tuples are immutable

```
a_tuple = (3, "car", 5)
print(a_tuple)

# Append element
a_tuple.append(True)
print(a_tuple)
```

```
(3, 'car', 5)
```

```
AttributeError: 'tuple'
object has no attribute
'append'
```

OVERVIEW OF SEQUENCE TYPES (SO FAR)

Type	mutable	item type
`list`	yes	mixed types
`tuple`	no	mixed types
`range`	no	integers
`str`	no	characters

Sequence operations are available according to mutability and type

COMMON SEQUENCE OPERATIONS

```
<bool> = el in s            # --> True/False
<bool> = el not in s        # --> True/False
s = s1 + s2                 # concatenate s1 and s2
s * n or n * s              # n times concatenation
```

Examples

```
print("a" in "Hallo")           # --> True/False
print(9 not in range(4, 10))    # --> True/False
print([100, 200] + [500, 20])   # concatenate s1 and s2
print(("cat", "dog") * 3)       # n times concatenation
```

```
True
False
[100, 200, 500, 20]
('cat', 'dog', 'cat', 'dog', 'cat', 'dog')
```

Note: concatenation does not work for ranges

COMMON SEQUENCE OPERATIONS

```python
el = s[i]                 # Select element i
s = s[start:end+1:step]   # Slicing
n = s.count[<el>]         # Count elements
i = s.index[<el>]         # index first el
```

For a tuple:

```python
s = ("Malta", "Corsica", "Lesvos", "Malta"); print(s)
print(f"Third element of s: { s[2] }")
print(f"Slice of s: { s[1:] }")
print(f"'Malta' appears: { s.count('Malta') } times")
print(f"First index of 'Malta': { s.index('Malta') }")
```

```
('Malta', 'Corsica', 'Lesvos', 'Malta')
Third element of s: Lesvos
Slice of s: ('Corsica', 'Lesvos', 'Malta')
'Malta' appears: 2 times
First index of 'Malta': 0
```

COMMON SEQUENCE OPERATIONS

```
el = s[i]                # Select element i
s = s[start:end+1:step]  # Slicing
n = s.count[<el>]        # Count elements
i = s.index[<el>]        # index first el
```

For a list:

```
s = ["Malta", "Corsica", "Lesvos", "Malta"]; print(s)
print(f"Third element of s: { s[2] }")
print(f"Slice of s: { s[1:] }")
print(f"'Malta' appears: { s.count('Malta') } times")
print(f"First index of 'Malta': { s.index('Malta') }")
```

```
['Malta', 'Corsica', 'Lesvos', 'Malta']
Third element of s: Lesvos
Slice of s: ['Corsica', 'Lesvos', 'Malta']
'Malta' appears: 2 times
First index of 'Malta': 0
```

SEQUENCE OPERATIONS: ONLY MUTABLE

List is the only mutable sequence (so far)

```
<list>[i] = <el>
<list>.remove(<el>)        # remove first <el>
<list>.insert(i, <el>)     # insert <el> at index i
<el> = <list>.pop(i)       # return <el> at i and remove
<list>.append(<el>)        # Or: <list> += <el>
<list>.extend(<iterable>)  # Or: <list> += <iterable>
<list>.sort()              # Sorts list in-place
<list>.reverse()           # Reverses list in-place
```

SEQUENCE OPERATIONS: ONLY MUTABLE

Examples:

```python
fruits = ["banana", "orange", "pear", "peach"]
nuts = ("almond", "walnut")

fruits.remove("banana")     # remove first <el>
fruits.insert(0, "mango")   # insert <el> at index i
el = fruits.pop(3)          # return <el> at i and remove
print(f"We popped {el}")
fruits.append("mandarin")   # Or: <list> += <el>
fruits.extend(nuts)         # Or: <list> += <iterable>
fruits.sort()               # Sorts list in-place
fruits.reverse()            # Reverses list in-place
print(fruits)
```

```
We popped peach
['walnut', 'pear', 'orange', 'mango', 'mandarin', 'almond']
```

SEQUENCE OPERATIONS: ALSO FOR IMMUTABLE

Items of immutable types cannot change ⇒ return value

```
<list> = sorted(<collection>)   # Returns sorted
<iter> = reversed(<list>)       # Returns reversed
```

Examples:

```
t = (3, 5, 7, 2, 1)
t_sort = sorted(t)    # Returns sorted
t_rev = reversed(t)   # Returns reversed
print(t)
print(t_sort)
print(list(t_rev))
```

```
(3, 5, 7, 2, 1)
[1, 2, 3, 5, 7]
[1, 2, 7, 5, 3]
```

STRINGS AS SPECIAL SEQUENCES

STRING SPECIFIC OPERATIONS

- Aside from the common sequence operations
- Specific to characters and text

```
print("   Some_extra spaces ")
print("   Some_extra spaces ".strip())
print("_Some text...".strip('.'))
```

```
   Some_extra spaces 
Some_extra spaces
_Some text
```

STRING SPECIFIC OPERATIONS

- Aside from the common sequence operations
- Specific to characters and text

```python
# Split and join strings
print("Split_on_underscore".split(sep="_"))
print( ".".join(["program", "exe"]) )
```

```
['Split', 'on', 'underscore']
program.exe
```

STRING SPECIFIC OPERATIONS

- Aside from the common sequence operations
- Specific to characters and text

```python
print("Ha" in "Hallo")
print("Ball".startswith("Bar"))
print("Bicycle".find("cyc"))
print("Bicycle".index("cyc"))
```

```
True
False
2
2
```

STRING SPECIFIC OPERATIONS

- Aside from the common sequence operations
- Specific to characters and text

```
print("GOOD morning".lower())
print("GOOD morning".upper())
print("GOOD morning".capitalize())
print("GOOD morning".title())
print("I like oranges".replace("oranges", "apples"))
```

```
good morning
GOOD MORNING
Good morning
Good Morning
I like apples
```

STRING SPECIFIC OPERATIONS

- Aside from the common sequence operations
- Specific to characters and text

```python
print(chr(68))  # Convert int to Unicode character.
print(ord("A")) # Convert Unicode character to int.
```

```
D
65
```

FOR LOOPS REVISITED

FOR LOOP OVER SEQUENCES: ALSO STRINGS

```python
# Print all letters of a string
for a in "Python 3.12":
  print(a)
```

```
P
y
t
h
o
n
 
3
.
1
2
```

FOR LOOP OVER SEQUENCES: ALSO STRINGS

```python
# Print all letters of a string reverse ordered
for a in reversed("Python 3.12"):
  print(a)
```

```
2
1
.
3
 
n
o
h
t
y
P
```

FOR LOOP OVER ELEMENTS WITH INDEX ?

- loop over the index from range with len() giving the length of the sequence
- select elements with the index

```python
# Print elements in order with their index
s = ["mouse", "chicken", "dog", "cow"]
for i in range(len(s)):
  print(f"The element with index: {i} = {s[i]}")
```

```
The element with index: 0 = mouse
The element with index: 1 = chicken
The element with index: 2 = dog
The element with index: 3 = cow
```

FOR LOOP OVER ELEMENTS WITH INDEX ?

- alternative: use enumerate
- lazy evaluated (just as range)
- gives both index and element

```
# Print elements in order with their index
s = ["mouse", "chicken", "dog", "cow"]
for i, el in enumerate(s):
  print(f"The element with index: {i} = {el}")
```

```
The element with index: 0 = mouse
The element with index: 1 = chicken
The element with index: 2 = dog
The element with index: 3 = cow
```

FLOW CONTROL: CONTINUE

continue skips a single iteration in **while** or **for** loop

```python
# Print elements in order with their index
numbers = [22, 20, 34, None, 25, 78]
for i, el in enumerate(numbers):
  if el is None:
    continue
  print(f"The element with index: {i} = {el}")
```

```
The element with index: 0 = 22
The element with index: 1 = 20
The element with index: 2 = 34
The element with index: 4 = 25
The element with index: 5 = 78
```

FLOW CONTROL: BREAK

break stops the current **while** or **for** loop

```python
# Print elements in order with their index
numbers = [22, 20, 34, None, 25, 78]
for i, el in enumerate(numbers):
  if el is None:
    break
  print(f"The element with index: {i} = {el}")
```

```
The element with index: 0 = 22
The element with index: 1 = 20
The element with index: 2 = 34
```

FLOW CONTROL: PASS

pass performs no action, purpose

- **clarify** inaction, provide **required** statement
- used for while, for, if, functions, classes, etc.

```python
numbers = [2, 10, 5, 3, 4, 7, 9, 3, 1]
count_high = 0; count_low = 0;
for el in numbers:
  if el > 8:
    count_high += 1
  elif el <= 4:
    count_low += 1
  else:
    pass
print(f"High: {count_high}, Low: {count_low}")
```

```
High: 2, Low: 5
```

LIST COMPREHENSIONS

WHEN A FOR LOOP IS CUMBERSOME

- to generate a simple list
- requires an index variable, can overwrite variable with same name

```python
# Print powers of 2 up to 1024
powers = []
for n in range(11):
  powers.append(2**n)
print(powers)
```

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

List comprehension

```python
powers = [2**n for n in range(11)]
print(powers)
```

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

WHEN A FOR LOOP IS CUMBERSOME

```python
cs = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if  x != y:
            cs.append((x, y))
print(cs)
```

```
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

List comprehension

```python
cs = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
print(cs)
```

```
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

ARRAYS

ARRAY DEFINITION

- arrays have a fixed length
- array elements have the same type
- We will use arrays from the Numpy library
- Optimized for numerical calculations

```python
# Load numpy for array
import numpy as np

a = np.array([1, 2, 4, 6, 5, 9])
print(a)
```

```
[1 2 4 6 5 9]
```

ARRAY INITIALIZATION

```python
import numpy as np

a = np.arange(15)                 # similar to range()
print(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

```python
a = np.linspace(0, 5, 11)   # linearly spaced interval
print(a)
```

```
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
```

```python
a = np.zeros((2, 3))              # 2D array with zeros
print(a)
```

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

ARRAY OPERATIONS

- similar to arithmetic/logic operations on numbers
- element-wise
- shapes need to be compatible

```python
import numpy as np

a = np.array([2.5, 3, 4])
b = np.array([1, 0.1, 10])
print(f"{a} + {b} = {a + b}")
print(f"{a} * {b} = {a * b}")
print(f"{a} / {b} = {a / b}")
```

```
[2.5 3.  4. ] + [ 1.   0.1 10. ] = [ 3.5  3.1 14. ]
[2.5 3.  4. ] * [ 1.   0.1 10. ] = [ 2.5  0.3 40. ]
[2.5 3.  4. ] / [ 1.   0.1 10. ] = [ 2.5 30.   0.4]
```

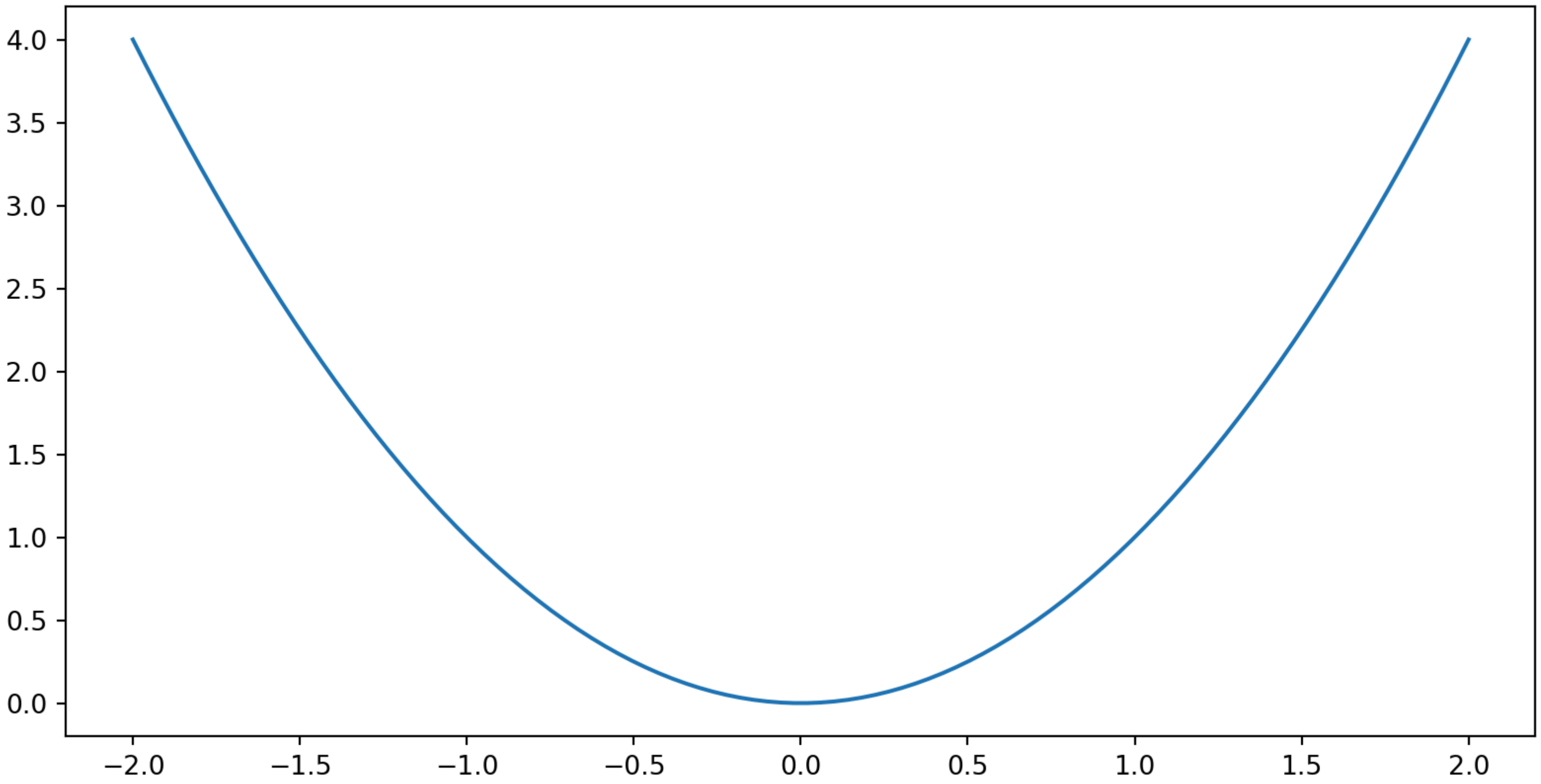
SIMPLE PLOTS

LINE PLOT

We will see more complex plots later on. If you want to look ahead: https://matplotlib.org/stable/users/explain/quick_start.html

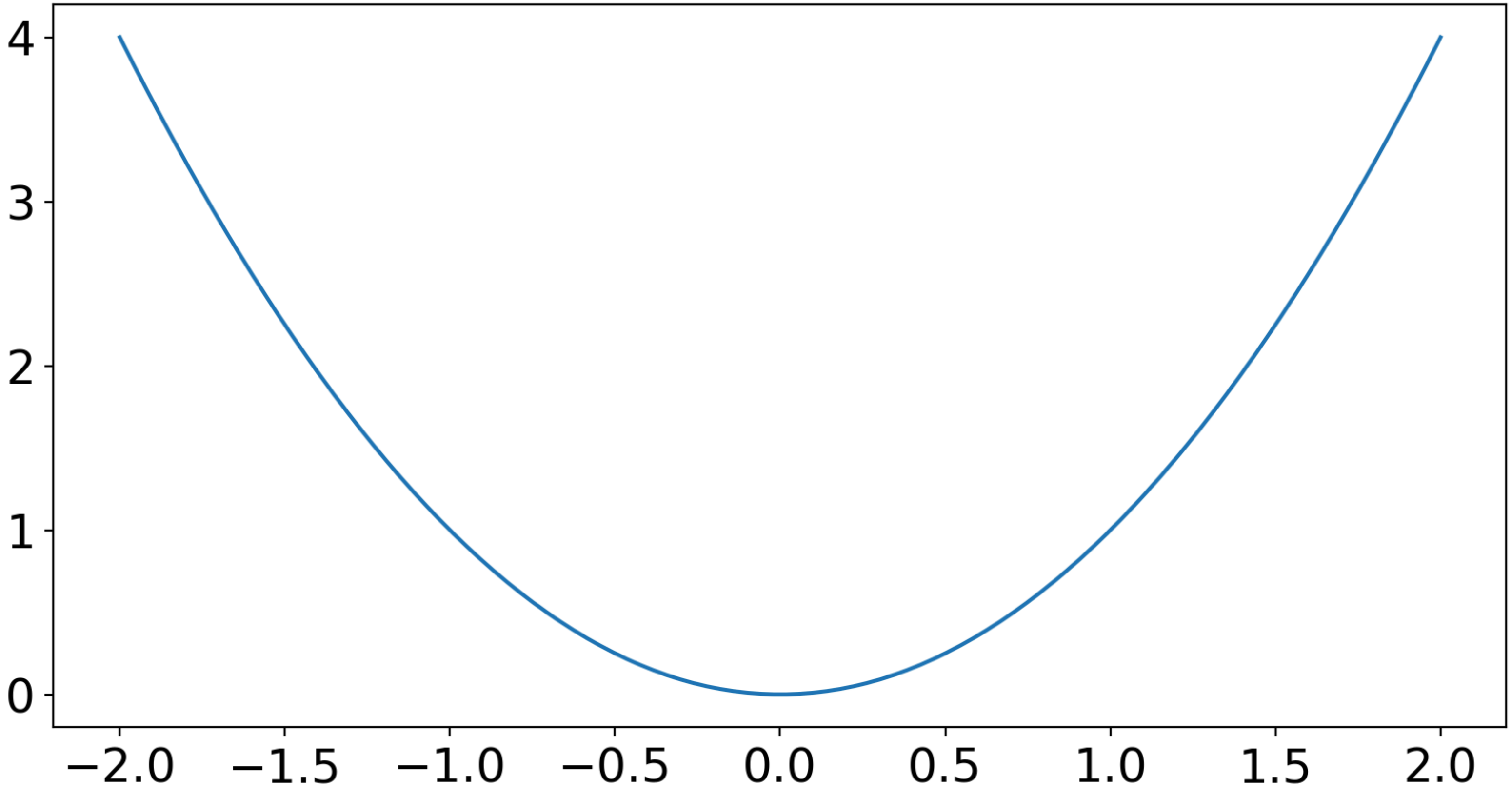
```python
# Import library for plotting and numerics
import numpy as np
import matplotlib.pyplot as plt

# Define x and y coordinates
x = np.linspace(-2, 2, 100)
y = x**2

# Plot a line between the coordinates
plt.plot(x, y);
```

LINE PLOT



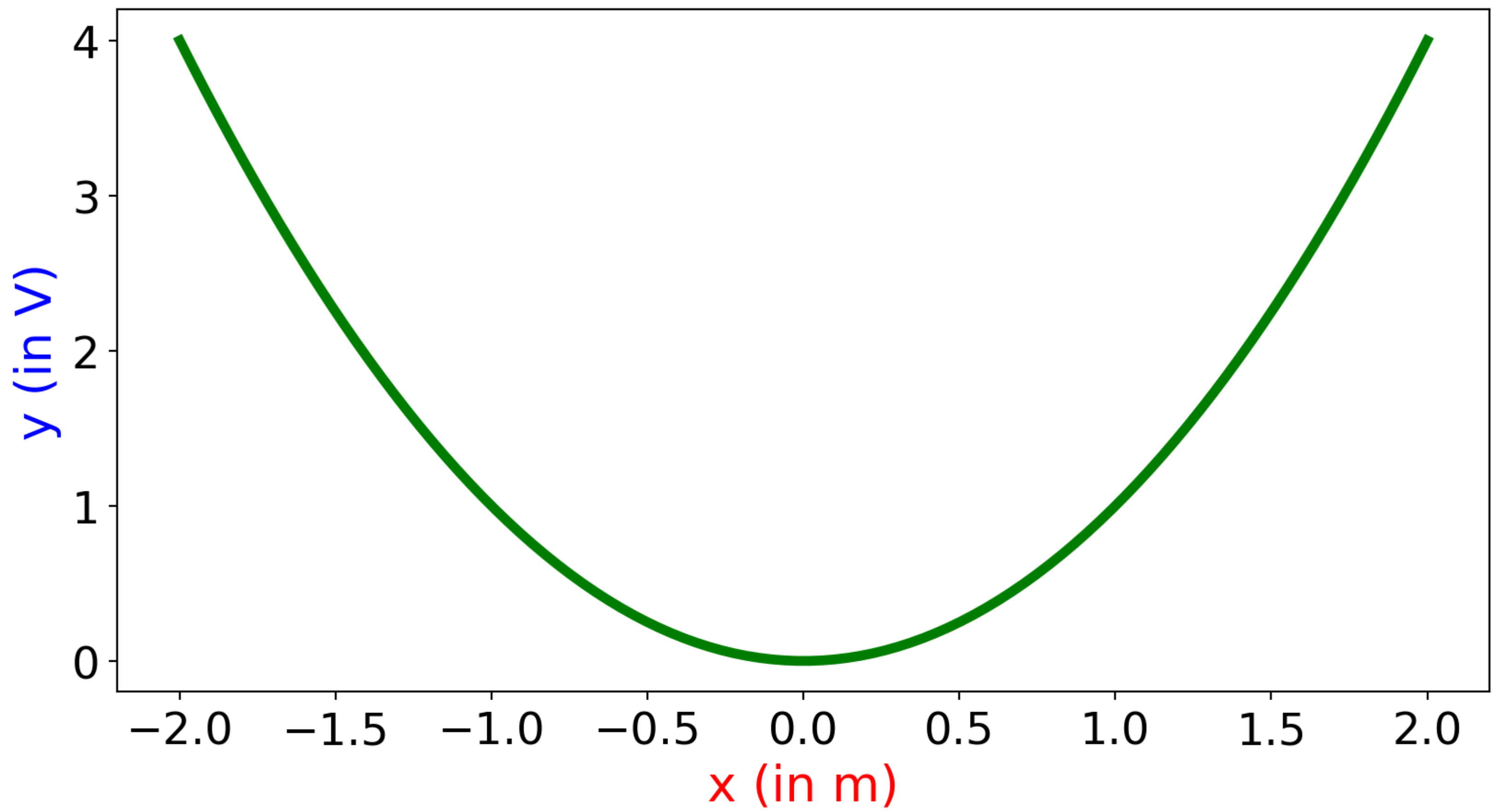
LINE PLOT

```python
# Change the font-size
import matplotlib as mpl
mpl.rcParams['font.size'] = 18
plt.plot(x, y);
```



LINE PLOT

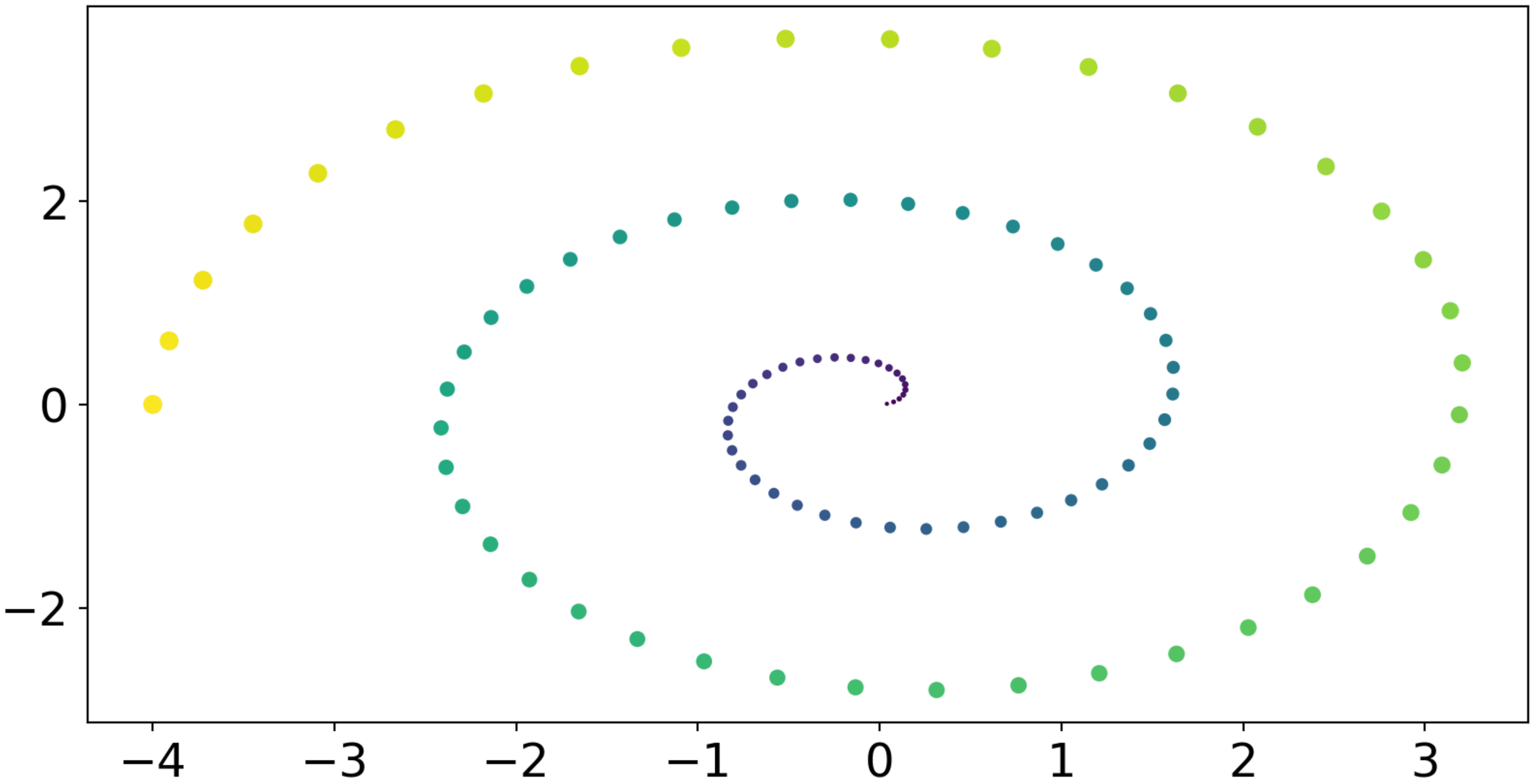
```python
# Add labels
plt.xlabel("x (in m)", fontsize=20, color='red')
plt.ylabel("y (in V)", fontsize=20, color='blue')
plt.plot(x, y, color='green', linewidth=4);
```


4
3
2
1
0
y (in V)
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x (in m)

SCATTER PLOT

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

# Create the coordinate of a spiral
angles = np.linspace(0, 5*np.pi, 100)
radii = np.linspace(0, 4, 100)
xs = radii * np.cos(angles)
ys = radii * np.sin(angles)

# Plot spiral points with increasing size and color-value
plt.scatter(xs, ys, 10*radii, radii/4);
```

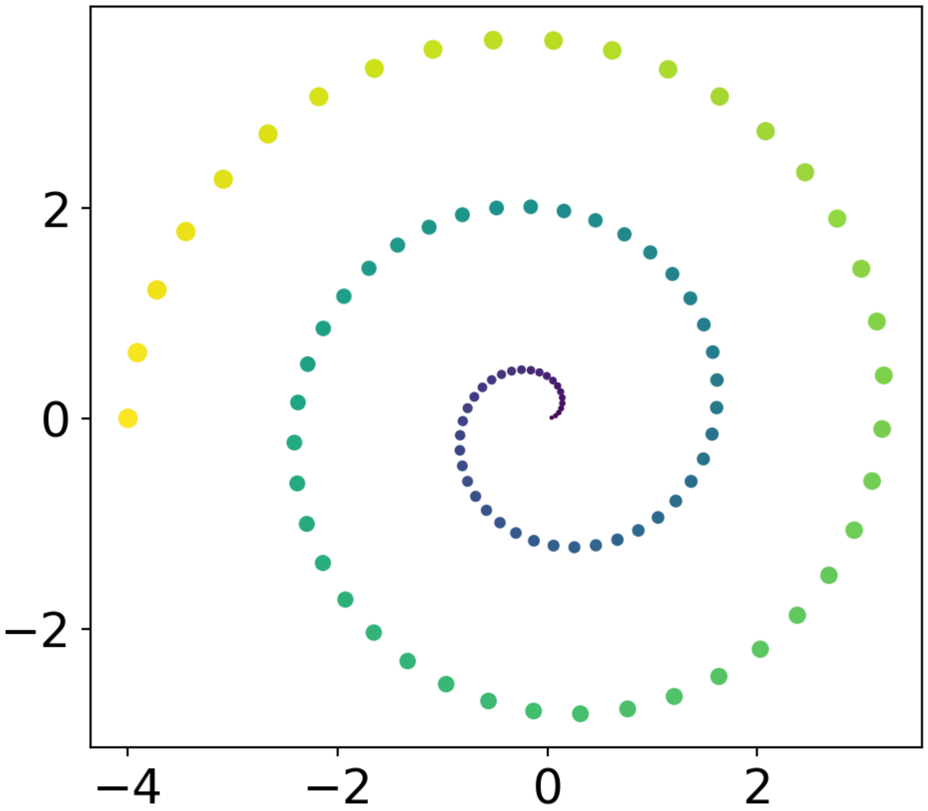
SCATTER PLOT



SCATTER PLOT

```
# Plot spiral points with increasing size and color-value
plt.scatter(xs, ys, 10*radii, radii/4)

# Set the axis aspect ratio to equal
ax = plt.gca()
ax.set_aspect("equal");
```

SCATTER PLOT



MORE ADVANCED LOOPING STRUCTURES

LOOPING MULTIPLE LISTS SIMULTANEOUSLY

- Loop can be over lists, arrays, sets, iterators, etc.
- There are different options to loop over two “lists” (say `a` and `b`) of same length:

1. **`while`** loop
2. **`for`** loop with **`range(len(a))`**
3. **`for`** loop with **`enumerate(a)`**
4. **`for`** loop with **`zip(a, b)`**
5. **list comprehension** with **`zip(a, b)`**

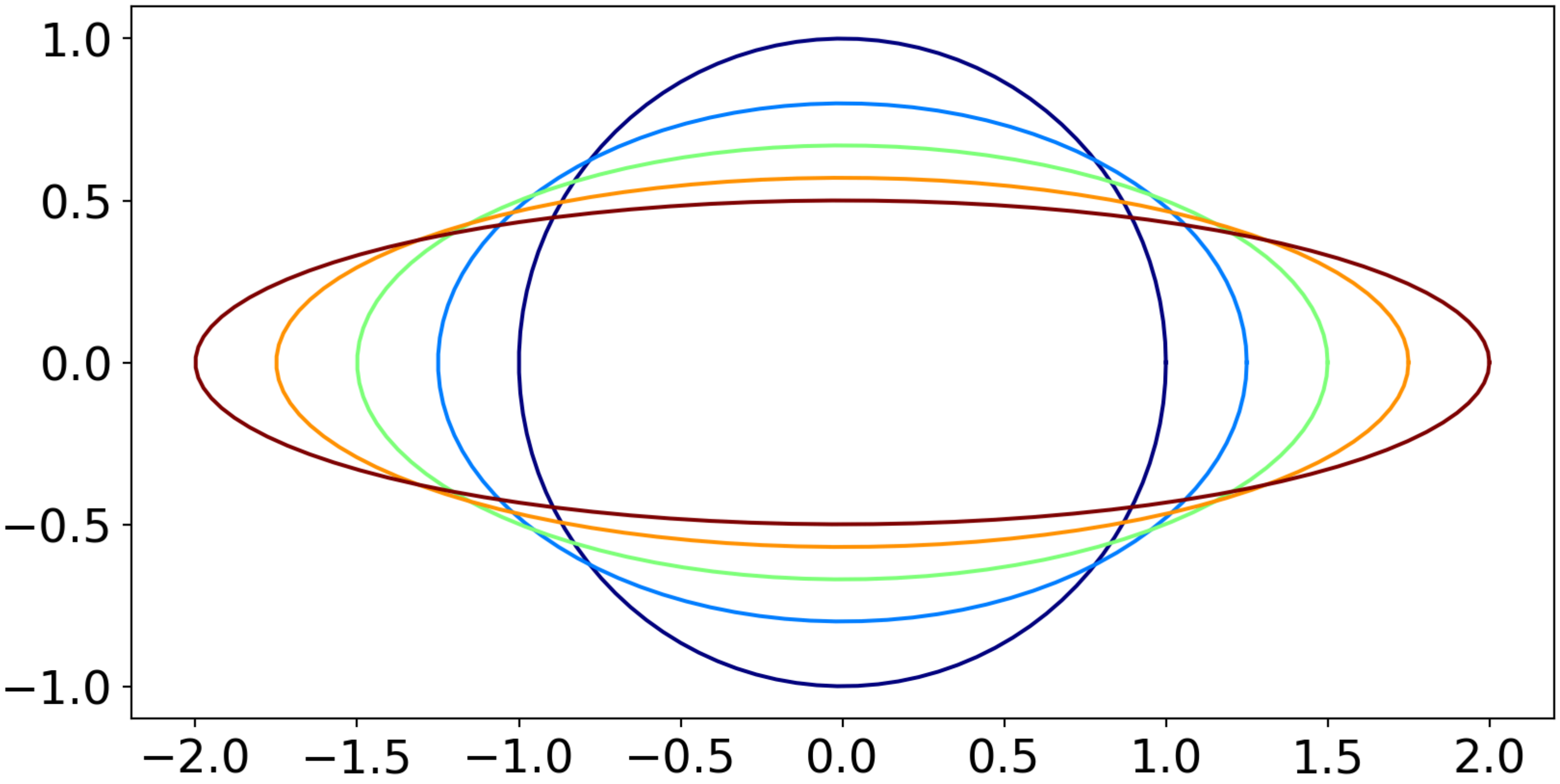
LOOPING MULTIPLE LISTS SIMULTANEOUSLY

- Exercise with ellipses with parameters: a and b

$$x = a\cos(t)$$
$$y = b\cos(t)$$

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

# the for loop allows to loop over a list or array:
n_curves = 5
t = np.linspace(0, 2*np.pi, 100)
a = np.linspace(1, 2, n_curves)
b = np.round(1 / a, 2)
```

LOOPING MULTIPLE LISTS SIMULTANEOUSLY



LOOPING MULTIPLE LISTS SIMULTANEOUSLY

- `len(a)` gives the length of the list

```
# Method 1 using while
i = 0
while i < len(a):
    x = a[i] * np.cos(t)
    y = b[i] * np.sin(t)
    i = i + 1
    # plt.plot(x, y)
```

LOOPING MULTIPLE LISTS SIMULTANEOUSLY

- `len(a)` gives the length of the list

```
# Method 1 using while
i = 0
while i < len(a):
  x = a[i] * np.cos(t)
  y = b[i] * np.sin(t)
  i = i + 1
  # plt.plot(x, y)
```

- `range(len(a))` gives numbers from 0 to `len(a) - 1`

```
# Method 2 using range
for i in range(len(a)):
  x = a[i] * np.cos(t)
  y = b[i] * np.sin(t)
```

LOOPING MULTIPLE LISTS SIMULTANEOUSLY

- Asymmetric solution with enumerate
- The parameters a and b are treated differently

```python
# Method 3 using enumerate
for i, ai in enumerate(a):
  x = ai * np.cos(t)
  y = b[i] * np.sin(t)
```

LOOPING MULTIPLE LISTS SIMULTANEOUSLY

- `zip(a,b)` merges lists a and b as iterator of tuples `(a[i],b[i])`
- An iterator is like a list, but lazy evaluated

```
print(list(zip(a, b)))
```

```
[(1.0, 1.0), (1.25, 0.8), (1.5, 0.67), (1.75, 0.57), (2.0,
0.5)]
```

LOOPING MULTIPLE LISTS SIMULTANEOUSLY

- `zip(a,b)` merges lists a and b as iterator of tuples `(a[i],b[i])`
- An iterator is like a list, but lazy evaluated

```
print(list(zip(a, b)))
```

```
[(1.0, 1.0), (1.25, 0.8), (1.5, 0.67), (1.75, 0.57), (2.0,
0.5)]
```

- usage in a `for` loop:

```
# Method 4 using zip()
for ai, bi in zip(a, b):
  x = ai * np.cos(t)
  y = bi * np.sin(t)
```

LOOPING MULTIPLE LISTS SIMULTANEOUSLY

- Does it make sense to us `enumerate()` ?
- `enumerate()` makes a “list” of tuples
- elements are the index and the tuples of `zip()`

```python
print(list(enumerate(zip(a, b))))
```

```
[(0, (1.0, 1.0)), (1, (1.25, 0.8)), (2, (1.5, 0.67)), (3,
(1.75, 0.57)), (4, (2.0, 0.5))]
```

```python
# Method 3 using enumerate
for i, tuple_i in enumerate(zip(a, b)):
  x = tuple_i[0] * np.cos(t)
  y = tuple_i[1] * np.sin(t)
```

This looks cumbersome !

LOOPING MULTIPLE LISTS SIMULTANEOUSLY

- Or combining **range()** and **zip()**?

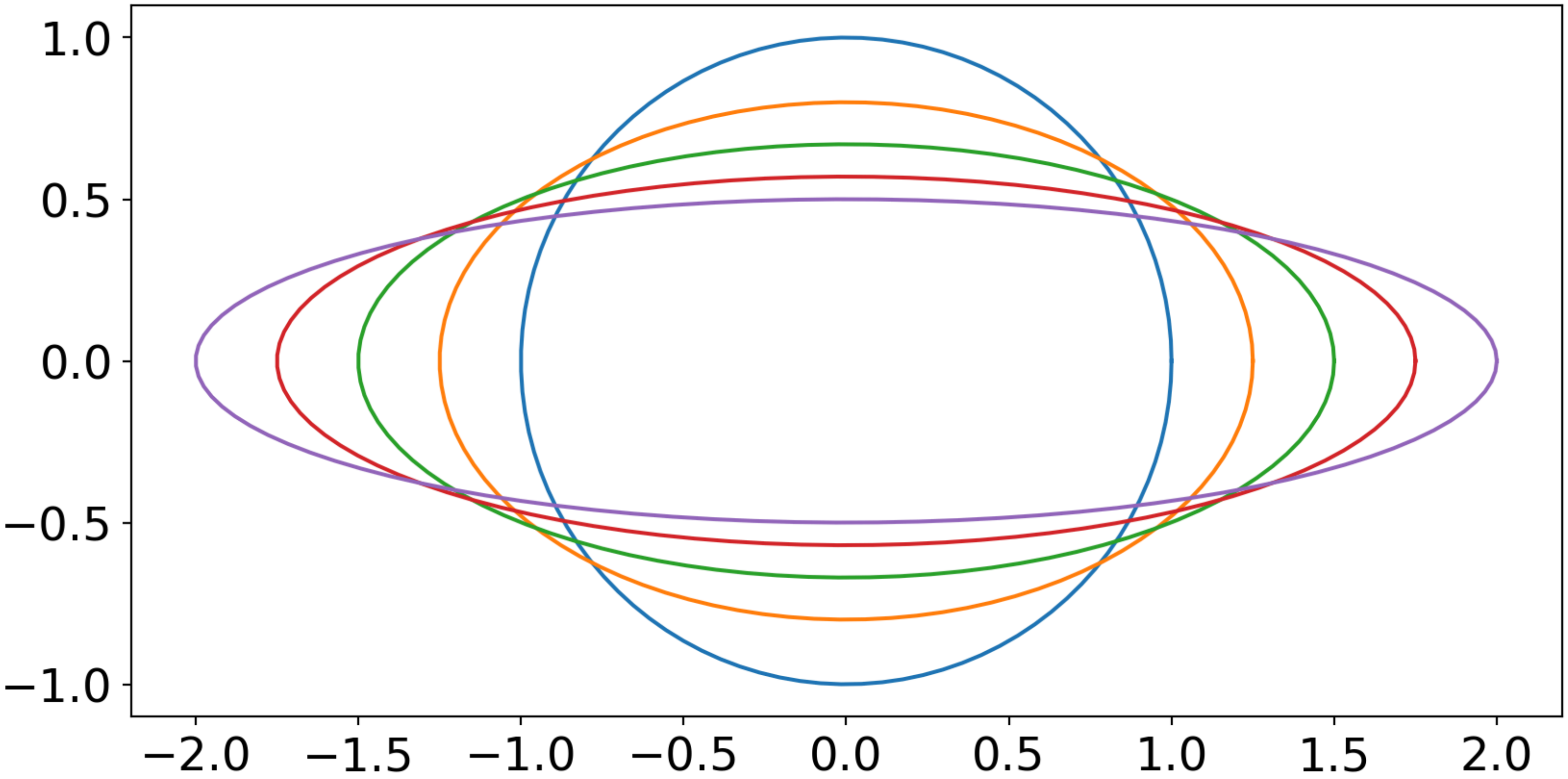
```
print(list(zip(range(len(a)), a, b)))
```

```
[(0, 1.0, 1.0), (1, 1.25, 0.8), (2, 1.5, 0.67), (3, 1.75,
0.57), (4, 2.0, 0.5)]
```

```
# Method 3 using enumerate
for i, ai, bi in zip(range(len(a)), a, b):
  x = ai * np.cos(t)
  y = bi * np.sin(t)
```

LOOPING MULTIPLE LISTS SIMULTANEOUSLY

```
# Method 5 using list comprehension with zip()
[plt.plot(ai*np.cos(t), bi*np.sin(t)) for ai, bi in zip(a
```


1.0
0.5
0.0
−0.5
−1.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0