Introduction to Programming with Python

Peter Georg

Disclaimer

History of Python

- First released in 1991.
- Releases are using semantic versioning (MAJOR.MINOR.PATCH).
- Major releases may introduce breaking changes, but basic principles are stable.
- Currently supported major release is 3.
- Optimized for fast development, readability, and portability.
- Includes pre-built solutions to reoccurring problems.

References

- Official Python 3 Documentation
- Official Python 3 Tutorial
- Style Guide for Python

Interpreter

- Computers can only execute machine specific language.
- Interpreter translates Python code to machine specific language in *real* time.
- Alternative approach: Compilers translate code before execution.
- **Advantage:** Interpreter handles machine specific details, i.e., (almost) all Python code runs on any machine assuming an interpreter exists.
- **Disadvantage:** Often suffers from lower performance compared to compiled languages.

Interpreter – Interactive Mode

- Interactive mode allows to directly feed interpreter with commands which will be executed instantly.
- Useful for testing and prototyping.
- Allows to use the power of Python for (small) daily tasks.
- Open in terminal by running `python3`.

Interpreter – Script

- Script: Text file that contains Python commands in a line-separated sequence.
- A script is executed by specifying it as an argument to the interpreter, i.e., `python3 <script>.py`.
- Additional arguments after the file are passed to the script.

First command

- The `print` command causes text to be output to the screen.
- Arguments are specified in parentheses `(...)`.
- Argument: Comma separated list of texts to be printed.
- Texts are enclosed in single (`'`) or double (`"`) quotes.
- By default every `print` command ends with a newline.
- By default output is sent to standard output.

Error messages

- Python reports when an error occurs.
- Report includes file, line number and copy of the line that caused the error and the type of the error (SyntaxError, NameError, ...).

Part I

Fundamentals

Variables

- Definition: A named symbol that stores a value.
- Value may be any kind of information, e.g., a number, text, a list of values, ...
- Its name may consist of `a-z`, `A-Z`, `_`, and `0-9` (not as the first character).
- Some names are reserved by Python for use as keywords.
- Variables are created by assigning a value to them, e.g., `x = 42`.
- Variables can be re-assigned to new values, e.g., `x = x + 1`.
- May be used in expressions, e.g., `x + 5`.
- May be used in statements, e.g., `print(x)`.

Recommendations

- A variable's name should self-explain its meaning.
  - The shortest possible name is not always the best!
- Use a single naming convention consistently.
  - Camel case: `thisIsAVariable`
  - Snake case: `this_is_a_variable`

User input

- The `input` command causes one line of text to be read from the keyboard.
- Arguments are specified in parentheses `(...)`.
- Optional argument *prompt*: Text to be shown before waiting for input to inform the user.
- Input is *returned* as a *string*.

  $\Rightarrow$ Type conversion might be necessary.
- **Example:** `x = input("Please enter a number: ")`
- **Warning:** User can enter non expected text!

Types

- Each expression, e.g., variable, has assigned information what it is, its *type*.
- *Operations* applied to a or multiple expressions are defined by their types.
- Numeric Types:
  - `int`: Integers, e.g., `x = 42`
  - `float`: Floating point numbers, e.g., `x = 3.14`
  - `complex`: Complex (floating point) numbers, e.g., `z = 4.2 + 2.4j`
- Sequence of characters, i.e, text:
  - `str`: *String*, e.g., `x = "This is a string"`
  - Prefix a special character with the *escape* character (\) to turn it into an ordinary character, e.g., `\"`.
- Find out the type of an expression using `type`, e.g., `print(type(2*x))`.

Arithmetic operators

Symbol	Function	Example
`+`	Addition	`x + y`
`-`	Subtraction	`x - y`
`*`	Multiplication	`x * y`
`/`	Division	`x / y`
`//`	Integer-Division	`x // y`
`%`	Modulo (Rest of a Division)	`x % y`
`**`	Power	`x ** y`
`-`	Negate	`-x`
`+`	Plus	`+x`
`abs`	Absolute value	`abs(x)`

- Parentheses and (mathematical) order of operations are obeyed.
- May use an arbitrary number of nested parentheses.

Shorthand

- Shorthand: `variable += expression`
- Abbreviation for `variable = variable + expression`
- Likewise: `-=`, `*=`, `/=`, ...
- Interpreter treats both versions (almost always) identically.
- **Advantage**: Removes a source of errors.

Types and Operations

- Remember: Operations are defined by the types they are applied to.
- Example: Integer vs. String
  - `1 + 2` evaluates to integer `3`
  - `"1" + "2"` evaluates string `"12"`
- Types may be incompatible, e.g., `"1" + 2` causes an error.

  $\Rightarrow$ Type conversion might be necessary.

Converting Types

- Expressions can be converted to other types.
- Use types, e.g., `float`, as functions: `float(1)` returns `1.0`
- Information might be lost during conversion.
  - `int(1.2)` return `1`
- Conversion may not always be possible:
  - `int("1")` returns `1`
  - `float("1.0")` returns `1.0`
  - `int("1.0")` causes an error

Comments

- Content of a line after a hash (#) is ignored by the interpreter.
- May be used to note information for the developer(s).
- May be used to switch off and on a statement for testing purposes.

Recommendations

- Use comments to explain your code if it is not obvious.
- Do not comment obvious code.
- Use self-explanatory names instead of writing comments.

String Formatting

- *Format Strings* allow to specify formatting by prefixing text by `f`.
- Expression to be included in formatted string in curly braces (`{}`).
- Example: `f"The solution of the equation is {x}"`.
- Optional format specifier after the expression separated by a colon (`:`).
- Format specifier may be used to set width, alignment, clipping, ...
- Possible format specifiers depend on the type of the expression.
- **Note:** Introduced in Python 3.6.

  Before that, the not so convenient `str.format()` method was used.

Floating point format specifiers

- Format specifier `f` (decimal notation) and `e` (scientific notation).
- May be prefixed by the number of decimal digits leading with a dot (`.`).
- May be prefixed by the minimum number of total characters.
- May be prefixed by a plus (`+`) or space to always include a sign or a space for negative values, respectively.
- Summary: `f"{x:{sign}{total}.{decimal}{format specifier}}"`
- Examples:
  - `f"{x:.4f}"`
  - `f"{x:8.4f}"`
  - `f"{x: 8.4f}"`
  - `f"{x:+.6e}"`

Types – Boolean

- Type: `bool`, Values: `True` and `False`
- Mostly: Result of comparisons, e.g., `1j**2 == -1` evaluates to `True`.
- Can be stored in a variable, e.g., `knowsComplex = 1j**2 == -1`.

Side notes

- Internally booleans are stored as integers `1` (`True`) and `0` (`False`).
- Integers converted to `bool` evaluate to `False` if `0` and `True` otherwise.

Comparison operators

Symbol	Function	Example
`==`	Equality	`x == y`
`!=`	Inequality	`x != y`
`<`	Less Than	`x < y`
`<=`	Less Equal	`x <= y`
`>`	Greater Than	`x > y`
`>=`	Greater Equal	`x >= y`

Logical operators

Symbol	**Function**	**Example**
`not`	Negation	`not x`
`and`	Conjunction	`x and y`
`or`	Disjunction	`x or y`

Example

```
print("Is x in the interval [0,1)?", x >= 0 and x < 1)
```

Conditional statement

- *If-Then-Else* statement.
- Execute code depending on a condition (boolean value).

If-Then Syntax

```
if condition:
    statements
```

If-Then-Else Syntax

```
if condition:
    statements
else:
    statements
```

- `statements` may be an arbitrary number of statements, but at least one.
- Conditional statements may be nested.

Indentation

- Indentation marks a block.
- Whitespace or tabulators.
- Same number of whitespace in each line of a block.

Recommendations

- Do **not** use tabulators.
- Use four whitespaces.
- Set your editor to expand tabulators to four whitespaces.

Example

```
temperature = float(input("Current temperature? "))

if (temperature > 20) and (temperature < 25):
    print("This is a pleasant temperature.")
else:
    print("It's either too hot or too cold.")

print("Temperature evaluation done.")
```

Conditional statement – Multiple Conditions

- *If-Then-ElseIf-Then-...-Else* statement.

```
if condition:
    statements
elif condition:
    statements
else:
    statements
```

- Conditions are checked one after the other beginning with the first.
- If several conditions are satisfied only the block of the first one is executed.
- May use an arbitrary number of `elif` blocks.
- May use an `else` block at the end.

Example

```python
temperature = float(input("Current temperature? "))

if temperature <= 20:
    print("It's too cold.")
elif temperature >= 25:
    print("It's too hot.")
else:
    print("This is a pleasant temperature.")

print("Temperature evaluation done.")
```

Recommendations

- Avoid redundant conditions.
- Avoid too many nested levels.
- Use logical operators to link conditions.
- Question your structures!

Conditional expression

- *Short* version of a conditional statement.
- Syntax: `expression if condition else expression`
- Example: `absoluteValueOfA = -a if a < 0 else a`

Recommendations

- Think of it as switching between two values, i.e., *one value or another*.
- Use only for simple conditionals.
- Otherwise use conditional statements.

Modules

- Collection of Python code.
- Python includes many standard modules, e.g., math and cmath.
- Later: Third-party modules.
- Modules allow code to be loaded only when actually needed.
- Use `import` to load a module, e.g., `import math`.
- *Objects* of a loaded module are not directly imported into your file, but are only available within a *namespace* which has the same name as the module.
- Syntax: `namespace.object`

Examples

- Module *math* contains many mathematical constants, e.g., π (`pi`) and e, and functions, e.g., sin, exp, log and $\sqrt{\ }$ (`sqrt`).
- Module *cmath* extends the support of math to complex values.

```
import math
import cmath

print( math.sqrt( math.pi))
print(cmath.sqrt(-math.pi))
```

Documentation for Python and standard modules

- docs.python.org/3
- Documentation is also distributed with the code.
- Access using `help`, e.g., `help(math)`, `help(math.sqrt)`.
- Recommendation: Use interpreter in interactive mode to read documentation.
- Third-party modules: Depends ...

Types – List

- Type: `list`, Values: Arbitrary many values of arbitrary type.
- Syntax: `variable = [element1, element2, ...]`
- Elements of a list are accessible via a continuous *index*.
- Syntax: `variable[index]`
- *Indices* start at 0.

```
values = [1, 3, 5.0, 1j, "Text", [1, 2, 3]]

print(values[5])
values[5] = 6

print(values)
```

Indices

- Indices have to be of type `int` or *slices*.
- Negative indices: The index $-n$ denotes the n-th element from the end of the list. The index -1 is used to access the last element.
- Slices are used to select a subsection of a list.
  - Syntax: `start:stop` or `start:stop:stride`
  - `start`, `stop`, and `stride` have to be integers.
  - `start` is included, `stop` is excluded.
  - Any part may be omitted.
  - Omitted start means from the beginning.
  - Omitted stop means until the (included) end of the list.

Examples

```
values = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(values[-1])        # 10
print(values[-2])        # 9
print(values[0:-1:2])    # [0, 2, 4, 6, 8]
print(values[2:-1:2])    # [2, 4, 6, 8]
print(values[::2])       # [0, 2, 4, 6, 8, 10]
print(values[:5])        # [0, 1, 2, 3, 4]
print(values[:])         # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Operations

- Concatenation of lists (`+`)
  - Example: `[5, 7] + [3]` returns `[5, 7, 3]`
  - Both operands have to be of type **`list`**.
- Repetition of lists (`*`)
  - Example: `[1, 3] * 2` returns `[1, 3, 1, 3]`
  - One operand has to be of type **`list`**, one of type **`int`**.
  - For negative integers the list is repeated zero times.

Methods

- A method is a *function* associated with an object.
- Syntax: `object.method(arguments)`
- `object`: An expression of any type, e.g., a variable of type `list`.
- `method`: A method available for the particular type.
- `arguments`: Comma separated list of expressions depending on the `method`.

List – Methods

- `append`: Append an element to a list.
  - Similar to `+=` but with *elements*, not lists.
  - Argument: Element to append.
  - Example: `listVariable.append(element)`
- `insert`: Insert an element into a list at specified position.
  - First argument: Index after which to insert the new element.
  - Second argument: Element to insert.
  - Elements after specified index are shifted back by one.
  - Example: `listVariable.insert(3, element)`

List – Methods

- `remove`: Remove an element from a list.
  - Argument: Element (**not** the index) to remove.
  - An error is raised if specified element is not in the list.
  - Only the first occurrence of the specified element in the list is removed.
  - Example: `listVariable.remove(element)`
- `pop`: Removes the element at specified position from the list.
  - Optional argument: Index of the element to be removed.
  - Otherwise the last element is removed.
  - Elements after specified index are shifted forward by one.
  - Example: `listVariable.pop(3)` or `listVariable.pop()`
  - Returns the removed element, e.g., `next = listVariable.pop()`.

List – Methods

- `reverse`: Reverse the order of the elements in a list.
  - Example: `listVariable.reverse()`
- `sort`: Sort the elements in a list by certain criteria.
  - Example: `listVariable.sort()`

Revisited: Assignment Operator

- We have learnt that the assignment operator `=` assigns a value to a variable.
- Can we use the assignment operator to create a copy of a variable?

```
a = [0, 1, 2, 3, 4, 5]

b = a
a[0] = -1

print(a)     # [-1, 1, 2, 3, 4, 5]
print(b)     # [-1, 1, 2, 3, 4, 5]
```

⇒ `b` is not created as a copy of `a`, but it *is* `a`.

Revisited: Variables

- Variables only store the *address* where the value is stored in memory.
- When accessing a variable Python automatically accesses the *referenced* value.
- Previous example: `a` and `b` both reference the same memory, i.e., values.
- But:

```python
a = 2

b = a
a = 4

print(a)    # 4
print(b)    # 2
```

⇒ `b` *is* **not** `a`.

Mutable and Immutable objects

- *Mutable* objects can be changed after construction.
- *Immutable* objects can **not** be changed after construction.
- Assigning a value to an immutable leads to construction of a new object.
- Objects of all types seen so far, except `list`, are immutable.

⇒ In line 4 of the code on the previous slide, `a` is a new object with no relationship to the previous `a` in line 1 other than the name.

List – Copying

- Non-nested lists:
  - Slicing creates a new list from the given at a new location in memory.

  ⇒ `listVariable[:]` creates a *true* copy.
  - **Note:** Elements themselves are not *true* copies, i.e., by assignment.
- Nested lists:
  - Module `copy`
  - `copy.copy`: Only copies first level of list, i.e., same as slicing.
  - `copy.deepcopy`: Recursively (*true*) copy each layer.

id and operator is vs. ==

- `id(variable)` returns the memory address of `variable`.
- `a == b` compares contents of variables `a` and `b`, e.g., their value.
- `a is b` compares the memory addresses of `a` and `b`.
- **Note:** `a is b` is equivalent to `id(a) == id(b)`.

Revisited: Types – String

- Text enclosed in single (`'`) or double (`"`) quotes.
- Internally: *Immutable list of characters*
- List operation are applicable:
  - Access individual characters using indices.
  - Create new (sub) strings using slices.
- Strings are **immutable**:
  - Access using indices is read-only.
  - When *modifying* a string a new string is created.

Types – Tuple

- **Immutable** type: `tuple`, Values: Arbitrary many values of arbitrary type.
- Syntax: `variable = (element1, element2, ...)`
- Essentially same as `list`, but immutable.
- No `append`, `pop`, `sort`, ...
- Addition of two tuples creates a new tuple.
- Often used as return value of functions.
  - Example: `divmod` returns quotient and remainder as tuple, e.g., `divmod(20, 3)` returns `(6, 2)`.
- Syntax for unpacking: `q, r = divmod(20,3)`

Types – Set

- **Mutable** type: `set`, Values: Arbitrary many unique values of arbitrary type.
- Syntax: `variable = {element1, element2, ...}`
- Can not access elements by index.
- Methods: `add`, `remove`, `clear`, `difference`, `intersection`, `union`, ...
- Addition operator is not defined.
- Logical operators (`and` and `or`) are defined.

Types – Frozenset

- Type `frozenset` is essentially the same as `set`, but **immutable**.
- Syntax: `variable = frozenset(element1, element2, ...)`

Types – Dictionaries

- **Mutable** type: `dict`, Values: Arbitrary many values of arbitrary type.
- Values are stored as key:value pairs.
- *Keys are used as indices.*
- Keys may (almost) be of arbitrary type.
- Syntax: `variable = {key1: element1, key2: element2, ...}`
- Access elements by key: `variable[key1]`
- Selected methods:
  - `keys`: Return (something like) a list of keys.
  - `values`: Return (something like) a list of values.
  - `items`: Return (something like) a list of `(key, value)` tuples.

Dictionaries – Example

```python
d = {"north": 12, "east": 9, "south": 6, "west": 3}

print(d["north"])     # 12
d["north"] = 0        # Write access

print(list(d.keys()))  # ["north", "east", "south", "west"]
print(list(d.values())) # [0, 9, 6, 3]

print(list(d.items())[0:2])     # [("north", 0), ("east", 9)]
```

Types – Ranges

- **Immutable** type: `range`, Values: Sequence of integers
- Syntax: `r = range(start, stop, stride)`
- Internally: Only values `start`, `stop` and `stride` are stored.
- The sequence is generated *on demand* when required.
- `start`, `stop` and `stride` have to be integers.
- `start` is optional with default value 0.
- `stride` is optional with default value 1.
- `stride` may be negative.

Ranges – Example

```
r = range(1, 8, 2)
print(r)                          # range(1, 8, 2)
print(list(r))                    # [1, 3, 5, 7]

r = range(1, 5)
print(list(r))                    # [1, 2, 3, 4]

r = range(0, 4)
print(list(r))                    # [0, 1, 2, 3]
r = range(4)
print(list(r))                    # [0, 1, 2, 3]

print(list(range(4-1, -1, -1)))   # [3, 2, 1, 0]
print(list(reversed(range(4))))   # [3, 2, 1, 0]
```

Containers: Special Functions and Operators

- `len`: Return number of elements in container.
  - `len([1, 2, 4])` returns `3`.
  - `len(range(5))` returns `5`.
- `in`: Return if (`bool`) a value is in a container.
  - `1 in [1, 2, 3]` returns `True`.
  - `[1] in [1, 2, 3]` returns `False`.
  - `1 in {1: "a", 2: "b"}` returns `True`.
  - `'a' in {1: "a", 2: "b"}` returns `False`.
  - `'a' in {1: "a", 2: "b"}.values()` returns `True`.

Containers: Special Functions and Operators

- `reversed`: Return a *copy* of the container with reversed order.
  - `reversed(range(5))` returns the sequence `4, 3, 2, 1, 0`.
  - Not supported by all containers.
- `sorted`: Return a *copy* of the container with sorted order.
  - Not supported by all containers.

Containers: Special Functions and Operators

- `zip`: Returns a sort of list with elements of containers *side-by-side* by creating tuples with elements from each container.
  - Length of created list is the length of the shortest container.

```
a = [1, 2, 3, 4]
b = ["a", "b", "c", "d"]
c = {0, 1, 2, 0}

print(list(zip(a,b,c)))
    # [(1, "a", 0), (2, "b", 1), (3, "c", 2)]
```

Loop statement – `for`

- *For each element in container do something* statement.

```
for element in container:
    statements
```

- `element` is *new variable* set to the elements of the container in sequence.
- `container` has to be an expression of type `list`, `tuple`, `range`, `dict`, ...
- `statements` may be an arbitrary number of statements, but at least one.
- Loop statements may be nested.
- Reminder: Blocks of code have to be indented.

Loop Examples – List

```
tasks = ["math", "physics", "housework"]

print("Your tasks today:")      # Your tasks today:
for task in tasks:
    print("*", task)
                                # * math
                                # * physics
                                # * housework

print()                         #
print("Get to work!")           # Get to work!
```

Loop Examples – Range

```python
print("Numbers 1 to 9 squared:")
                                  # Numbers 1 to 9 squared:
for i in range(1, 10):
    print(f"{i}^2 = {i**2:2}")
                                  # 1^2 =  1
                                  # 2^2 =  4
                                  # 3^2 =  9
                                  # 4^2 = 16
                                  # 5^2 = 25
                                  # 6^2 = 36
                                  # 7^2 = 49
                                  # 8^2 = 64
                                  # 9^2 = 81
```

Loop Examples – Tuples

```python
import math

vectors = [(1, 1), (4, 7), (-1, 2)]

for v in vectors:
    print("vector", v, "has length", f"{math.hypot(v[0], v[1]):.4f}")

# vector (1, 1) has length 1.4142
# vector (4, 7) has length 8.0623
# vector (-1, 2) has length 2.2361
```

Loop Examples – Unpacking Tuples

```python
books = [("Frank Herbert", "Dune"),
  ("Douglas Adams", "The Hitchhikers Guide To The Galaxy"),
  ("Randall Munroe", "What If")]

print("You might want to read:")
for author, title in books:
    print("*", title, "by", author)

# You might want to read:
# * Dune by Frank Herbert
# * The Hitchhikers Guide To The Galaxy by Douglas Adams
# * What If by Randall Munroe
```

Loop Examples – Dictionaries

```
d = {"north": 12, "east": 9, "south": 6, "west": 3}

for key, value in d.items():
    print(f"{key:5}: {value:2}")

# north: 12
# east :  9
# south:  6
# west :  3
```

Loop – `enumerate`

`enumerate` generates a list of tuples holding indices and elements of a container.

```python
tasks = ["math", "physics", "housework"]

print("Your tasks today:")      # Your tasks today:
for i, task in enumerate(tasks):
    print(f"{i+1}. {task}")
                                # 1. math
                                # 2. physics
                                # 3. housework

print()                         #
print("Get to work!")           # Get to work!
```

Loop – `zip`

`zip` generates a list of tuples holding elements of each container at the same position.

```
data1 = [1.7, 2.2, -4.1]
data2 = [1.8, 2.0, -3.8]
print("Difference in datasets:")
for i, t in enumerate(zip(data1, data2)):
    print(f"Datapoint {i}: {t} differs by: {t[0] - t[1]:.2f}
        ")

# Datapoint 0: (1.7, 1.8) differs by: -0.10
# Datapoint 1: (2.2, 2.0) differs by: 0.20
# Datapoint 2: (-4.1, -3.8) differs by: -0.30
```

List comprehension

- Short version to create a container from contents of another container:

```
listVar = [        expr for x in container if condition]
setVar  = {        expr for x in container if condition}
dictVar = {kexpr: vexpr for x in container if condition}
```

- `x`: Helper variable set to the element from `container` in sequence.
- `container` has to be an expression of type `list`, `tuple`, `range`, `dict`, ...
- `expr`: Arbitrary expression decribing the elements of the new container.
- Only elements for which `condition` is `True` are included.
- `if` condition is optional. Arbitrary many `if condition`s may be appended.
- `x` is accessible in all expressions, e.g., in `expr` and `condition`.

Examples

```
import math

N = 3
eValues = [math.exp(x) for x in range(N)]
# [1.0, 2.718281828459045, 7.38905609893065]
```

```
[[3*row + col+1 for col in range(3)] for row in range(3)]
# [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
M = 4
N = 17
[x for x in range(M,N) if x % 5 != 0 if x % 7 != 0]
# [4, 6, 8, 9, 11, 12, 13, 16]
```

Loop statement – `while`

- *As long as condition is satisfied repeat something* statement.

```
while condition:
    statements
```

- `condition` is checked before each repetition.
- `statements` may be an arbitrary number of statements, but at least one.
- Loop statements may be nested.
- Reminder: Blocks of code have to be indented.
- Warning: Possible to create *infinite* loops.

Example

```python
capital = float(input("Your seed capital: "))
interest = float(input("Interest rate: "))
limit = float(input("Your savings goal: "))

years = 0
while capital < limit:
    capital *= 1 + interest
    years += 1

print(f"After {years} years you've reached your goal.")
```

Altering the flux of loops – `break`

- `break` may be used to exit loops at any time.
- Often used for secondary condition within the loop.
- In case of nested loops it exits the *current* level.
- Hint: Use flags to leave multiple levels.
- Example:

```
import math

for x in range(2,10):           # 2 7.38905609893065
    if math.exp(x) > 1000:      # 3 20.085536923187668
        break                   # 4 54.598150033144236
    print(x, math.exp(x))       # 5 148.4131591025766
                                # 6 403.4287934927351
```

Altering the flux of loops – `continue`

- `continue` may be used to skip the rest of the current iteration of the loop.
- Often used to skip single elements.
- In case of nested loops it skip the rest of the *current* level.
- Example:

```
N=10
S=4

for x in range(-N,N+1,S):         # -0.1000
    if x == 0:                    # -0.1667
        continue                  # -0.5000
    print(f"{1/x:+.4f}")          # +0.5000
                                  # +0.1667
                                  # +0.1000
```

Flux of loops – `else`

- May attach an `else` block to loops.
- `else` block is executed if the loop has **not** been exited using `break`.
- Idea: *If not applicable*
  - `for`: Trigger element not in container (*if trigger then break*)
  - `while`: Trigger condition never met (*if trigger then break*)

Example

```python
capital = float(input("Your seed capital: "))
interest = float(input("Interest rate: "))
limit = float(input("Your savings goal: "))

years = 0
while capital < limit:
    if interest <= 0:
        print("Goal can never be reached.")
        break

    capital *= 1 + interest
    years += 1
else:
    print(f"After {years} years you've reached your goal.")
```

Which loop statement should be used?

As with many quesitons in programming there is no one-size-fits-all answer.
A sensible guideline is to use

- `for` with containers and
- `while` with conditions.

Note: Sometimes it is sensible to loop over containers using a condition, especially when changing the size of the container within the loop.

Functions

- A *function* is used to combine multiple recurring steps.
- Syntax:

```
def functionName(parameters):
    statements

functionName(arguments)
```

- `functionName`: Name of the function to be defined.
- `arguments`: Objects passed to the function.
- `parameters`: Comma separated list of parameters bound to `arguments`.
- `statements` may be an arbitrary number of statements, but at least one.

Functions – `return`

- `return returnValue`: Exit function, i.e., skipping any further statements, by returning to where it was called and *return* `returnValue`.
- `return`: `returnValue` is optional. When ommited returns `None`.
- Implicit `return` after last statement in function block, e.g., `statements`.
- A function block may contain arbitrary many `return` statments.

```python
def polynomial(x):
    return x**2 - 5*x + 3

print(polynomial(5))     # 3
```

Functions

- Desire: A function call shall have no side effects.
- But: A function may need to use auxiliary objects.
  - Objects may stay in memory.
  - Objects may overwrite other objects if symbol name already used.
- Solution: Functions have their own *scope*.
  - Function has its own set of *local* objects.
  - Function as a closed box.

Scopes

- *Lifetime of a symbol*
- Each symbol only exists in a given range of code.
- Objects are deleted once *out of scope*, e.g., *local* variables of a function are deleted when exiting the function.
- May temporarily overwrite another symbol in a nested scope.
- *Global* (or *module*) scope: Outermost scope.

Scopes

```python
def func():
    x = 1
    y = 1
    print(x, y)     # 1 1

x = 2
print(x)            # 2

func()

print(x)            # 2
print(y)            # Error: y not defined in this scope.
```

Functions – Passing arguments

- Idea: *Pass* information into a function.
- Parameters: Local variables of the function.
- Parameters are initialized by a *copy* of the specified arguments.

```python
def func(x, y):
    print(x, y)             # 2 4
    x, y = 1, 1
    print(x, y)             # 1 1

x = 2
func(x, x + 2)
print(x)                    # 2
```

Functions – Kind of arguments

- *Keyword* argument:
  - An argument preceded by an identifier in a function call, e.g.:
    `f(x=4, y=2)`
  - Arguments passed as values in a dictionary preceded by `**`, e.g.:
    `f(**{x: 4, y: 2})`
- *Positional* argument:
  - An argument that is not a keyword argument, e.g.:
    `f(4, 2)`
  - Arguments passed as values in an *iterable* preceded by `*`, e.g.:
    `f(*(4, 2))`
- Positional arguments may only be used before any keyword argument in an argument list.

Functions – Kind of parameters

- *positional-or-keyword* parameter (default):
  - Specifies an argument that can be passed positionally or by keyword.
  - Syntax: `def f(x, y)`
- *positional-only* parameter:
  - Specifies an argument that can only be passed positionally.
  - Use `/` in parameter list to mark end of positional only parameters.
  - Syntax: `def f(p1, p2, /, x, y)`
- *keyword-only* parameter:
  - Specifies an argument that can only be passed by keyword.
  - Use `*` in parameter list to mark beginning of keyword only parameters.
  - Syntax: `def f(x, y, *, k1, k2)`

Functions – Kind of parameters – Variadic

- *var-positional* parameter:
  - Specifies an arbitrary count of positional arguments.
  - Syntax: `def f(*args)`
  - `args` is of type `tuple`.
- *var-keyword* parameter:
  - Specifies an arbitrary count of keyword arguments.
  - Syntax: `def f(**kwargs)`
  - `kwargs` is of type `dict` with keys of type `str`.

Optional arguments – Default values

- May set a default value for any non-variadic parameter.
- Syntax: `def f(x, y, n = 1000)`
- Parameters with default values may be omitted in function call.
- **Note:** When using positional arguments with default values only arguments from the end backwards may be omitted.

Lambda expression

- Small anonymous functions.
- Syntax: `lambda args : expression(args)`
- Equivalent:

```
def ???(args):
    return expression(args)
```

- Lambdas may be called like regular functions.
- Rules of regular functions apply.

Example

```
stuffToSort = [-1, 2, -5, 4, 3]

stuffToSort.sort(key = lambda element : abs(element))

print(stuffToSort)     # [-1, 2, 3, 4, -5]
```

What order for the parameters?

- One must be able to distinguish which argument belongs to which parameter.

Suggestions

- positional-only parameters shall be specified first.
- keyword-only parameters shall be specified last.
- Default values shall be specified for parameters from the end backwards.
- Variadic parameters shall be specified after non-varidic ones of their kind.
- For positonal-or-keyword follow either the suggestion for positional-only or keyword-only parameters depending on preferred way to pass the arguments.
- Do not use positional-only or keyword-only parameters.

Example – `print`

- Function signature:

  `print(*args, sep=' ', end='\n', file=None, flush=False)`
- `args`: Objects to be printed.
- `sep`: Separator (`str`) between each object to be printed.
- `end`: String to be printed after all objects.
- `file`: Target to print to. Default: `sys.stdout`
- `flush`: Flush stream afterwards.

Function generators

- Functions may be nested.
- A function may return an inner function.
- A new copy of the inner function is created for each call of the outer function.
- Parameters if the outer function are part of the copy of the inner function.
- Allows us to dynamically generate new functions.

Example – Numerical differentiation

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

```python
import math

def derivative(f, epsilon = 1e-7):
    def result(x):
        return (f(x + epsilon) - f(x)) / epsilon
    return result

dsin = derivative(math.sin)
dcos = derivative(math.cos)

print(f"{dsin(math.pi): .4f}") # -1.000
print(f"{dcos(math.pi): .4f}") #  0.000
```

Recursive Function

- Functions may call other functions and, in particular, may call themselves.
- Each call creates a new set of local variables.
- Useful for solving *self-similar* problems, i.e., for problems that can be decomposed into sub-problems where:
  - Sub-problems are of the same form as the main problem.
  - The size of the sub-problems decrease.
  - There exists at least on trivial case.
  - The problem decays into a set of trivial cases.

Example – Sum over elements in a list

Iterative approach: Calculate sum of all elements, e.g., by using `sum`.

Recursive approach: The sum over all elements in a list is equal to its first element plus the sum over the remaining elements.

```
def recSum(data):
    if len(data) == 1:
        return data[0]
    else:
        return data[0] + recSum(data[1:])

print(recSum(list(range(4))))    # 6
```

Remarks

- Maximum recursion depth (by default): 1000
- Additional overhead $\Rightarrow$ (Possibly) slow execution!
- *Some* problems can be solved extremely efficiently using recursion.
- Theoretically every recursive algorithm can be implemented iteratively.
- In practice this may prove almost impossible.

Revisited: Functions – Passing arguments

- Other programming languages: *pass-by-value* and *pass-by-reference*.
- *pass-by-value*: Arguments are copied into function.
- *pass-by-reference*: References to arguments are passed to function.
- Python *binds* parameters to passed arguments.

⇒ Immutable and Mutable objects behave differently.

Passing immutable objects

```
def f(x):
    x = 2
    print(x)     # 2

x = 1
f(x)
print(x)         # 1
```

Passing mutable objects

```python
def f(data):
    data.append(1)
    data = data + [2]
    data.append(3)
    print(data)                 # [1, 2, 3]

data = []
f(data)
print(data)                     # [1]
```

Note: `data += [2]` behaves like `data.append(2)`.

Accessing outer scope variables

- Symbols defined in an inner scope are not visible in an outer scope.
- Symbols defined in an outer scope are visible in an inner scope.
- *Can not see into a scope, but out of a scope.*

⇒ Functions have access to *nonlocal* objects.

- Type of first access in a function defines behavior:
  - Read access: Nonlocal variable.
    Symbol stands for the variable defined in the outer scope.
  - Write access: Local variable.
    Symbol stands for the new object defined in the inner scope.

Example

```python
def f(z):
    y = 1
    print(x, y, z)     # 0 1 1

x = 0
y = 0
z = 0

f(1)

print(x, y, z)         # 0 0 0
```

Example – Erroneous code

```
def f(z):
    print(x, y, z)
    # UnboundLocalError: local variable 'y' referenced before assignment
    y = 1

x = 0
y = 0
z = 0

f(1)

print(x, y, z)
```

Keyword `global`

- Use `global` to explicitly bind a symbol to the global symbol.
- Syntax: `global variableName`
- Allows us to change variables of global (or module) scope.

Strong Recommendation

Do **not** use `global`!

Example

```python
def f():
    global x
    print(x)      # 0
    x = 1
    print(x)      # 1

x = 0

print(x)          # 0
f()
print(x)          # 1
```

Keyword `nonlocal`

- Essentially the same as `global`, but for nested functions.
- Binds symbol to symbol of outer scope, but not global (or module) scope.

```
def f():
    x = 0
    def g():
        nonlocal x
        x += 1
```

Recommendation

Do not use `nonlocal`.

Recommendations

- A function shall do one thing and one thing only.

  ⇒ *Single-responsibility principle* (SRP)

- Only use arguments to pass data into functions.
- Only use return to pass data out of functions.
- Exception: Update of mutable objects, e.g., `updateState(variable)`.
- Use self-explanatory function names.
- Use self-explanatory parameter names.

Opening files

- We can *open* a file for reading from or writing to it.
- Syntax: `handle = open(filename, mode)`
- `filename`: Relative or absoulte filename, e.g.:
  - `"data.txt"`: File in current working directory.
  - `"../data.txt"`: File one level above current working directory.
  - `"/home/gep21271/data.txt"`: Absolute filename (Linux).
  - `"C:/Users/UserName/data.txt"`: Absolute filename (Windows).

  Note: Separator is a forward slash (`/`), not the backslash (`\`).
- `mode`: A string combination describing the way in which the file will be used.
- Further parameters with default arguments.

Opening files – Handle

- `handle`: Used to access (and manipulate) the file `filename`.
- `handle`: Holds the current *position* in a file.
- *Position* in a file is measured in bytes from the beginning of the file.
- `handle.tell()`: Returns the current file position in a file stream.
- `handle.seek(offset, whence)`:
  - Changes position of the `handle` in file.
  - Position is computed by adding `offset` to a reference point (`whence`):
    - `0`: Reference point at the beginning of the file.
    - `1`: Reference point at the current file position.
    - `2`: Reference point at the end of the file.

  Default argument: `whence` = `0`

Opening files – Access

Mode	Access	Position	
`"r"`	Read	Beginning	Error if file does not exist.
`"r+"`	Read & Write	Beginning	Error if file does not exist.
`"w"`	Write	Beginning	Truncation of existing file.
`"w+"`	Read & Write	Beginning	Truncation of existing file.
`"x"`	Write	Beginning	Error if file does exist.
`"x+"`	Read & Write	Beginning	Error if file does exist.
`"a"`	Write	End	Writes always append to the file.
`"a+"`	Read & Write	End	Writes always append to the file.

Opening files – `mode`

- Combination of file access and (optionally) binary or text mode.
- Binary mode (`"b"`): Read and write content as bytes.
- Text mode (`"t"`): Read and write content as strings.
- Text mode is assumed if not specified otherwise.
- Default file access: `"r"`.

Text or binary mode?

- If the file should also be read by humans, use text mode.
- If the file should only be read by programs, use binary mode.

Writing to files

- May write to a file opened with write access.
- Syntax: `handle.write(expr)`
- `expr` has to be of type `str` (text mode) or `bytes` (binary mode).
- `expr` is written into the file after the current position of `handle`.
- Position of `handle` is moved along while writing.
- Afterwards position is behind the last character or byte written.

Reading from files

- May read from a file opened with read access.
- Syntax: `handle.read()`
- Retuns a `str`ing or `bytes`.
- Optional positional argument:
  - Maximum number of characters or bytes to be read.
  - May read less characters or bytes if file is shorter.
  - Position of `handle` is moved along while reading.
  - Afterwards position is in front of the first unread character or byte.
  - Default: Read entire file.
- Attempting to read past the end of the file will raise an error.

Closing files

- The Operating System requires us to close a file handle once done.
- Syntax: `handle.close()`
- If forgotten the Python interpreter might close the file eventually.
- This may result in a noticeable delay.

Anticipation: `with` statement

- May use a `with` statement to automatically close files:

```
with open(filename, mode) as handle:
    statements

# File is automatically closed after the with statement.
```

Reading line(s) from files

- Method `readline` reads an entire line.
- Method `readlines` reads all lines as a `list`.
- Optional positional argument: Maximum size to be read in characters/bytes.
- Line breaks are included in the read content.
- File handles are iterable, e.g., can use `for` to read line by line.
- Note: Uses `readline` internally.

```python
with open("output.txt", "r") as handle:
    for line in handle:
        print(line)
```

Excursion: Types – `bytes`

- Type representing a section of memory.
- May be used to write memory content as is into binary files.
- May be generated from many objects.
- May be converted into many data types.
  - Requires to specify further information, e.g., endianness.

Excursion: Types – bytes

```
x = 65535
bX = x.to_bytes(2, byteorder="little", signed=False)
print(bX)     # b'\xff\xff'

handle = open("output.dat", "wb")
handle.write(bX)
handle.close()

handle = open("output.dat", "rb")
bX = handle.read(2)
handle.close()

x = int.from_bytes(bX, byteorder="little", signed=True)
print(x)      # -1
```

Recommendations

- Do not mix read and write acces, i.e., use only `"r"`, `"w"` or `"a"`.
- Prefer relative paths to absoulte paths.
- Avoid using `seek` and `tell`.
- Write `open` and `close` in one step.
- Use `with` statements.

Serialization of Python objects – `pickle`

- Module `pickle` may be used to *serialize* **any** object.
- Useful to write/read **any** object to/from a file (in binary mode).
- Write: `pickle.dump(object, handle)`
- Read: `object = pickle.load(handle)`
- Python specific format.

  Warning: Poses a huge security issue as loaded data might contain anything.

  Note: From the file itself it is hardly possible to see what is loaded.

  Recommendation: Only load pickle files that you really trust!

Portable archives – JSON

- *JavaScript Object Notation*
- Widely used data format supported in many programming languages.
- Data is stored as humand-readable text.
- Only supportes 8 data types and compositions thereof:
  `int`, `float`, `str`, `bool`, `list`, `tuple`, `dict`, `None`.
- Other data types have to be deconstructed manually.
- Write: `json.dump(object, handle)`
- Read: `object = json.load(handle)`
- **Note:** File `handle` has to be opened in text mode.
- **Documentation:** docs.python.org/3/library/json

```python
import json

complexNumber = -0.3 + 4.1j
dictionary = {1: "one", 2: "two", 4: ["list", "strings"]}

with open("archive.json", "w") as handle:
    json.dump({
        "real": complexNumber.real,
        "imag": complexNumber.imag,
        "dict": dictionary
    }, handle)

with open("archive.json", "r") as handle:
    jObject = json.load(handle)

print(complex(jObject["real"], jObject["imag"]))
print(jObject["dict"])
```

Tables – CSV

- *Comma Separated Values*
- Widely supported format by many spreadsheet programs.
- String, integer and floating point number in rows and columns.
- Columns separated by an arbitrary separated character, usually a comma.
- Strings usually in *string delimiters*, usually double quotes.

Tables – CSV

- `csv.reader`: *Class* that parses a CSV file with a given set of rules.
- Arguments when creating an object of type `csv.reader`:
  - `handle`: File opened in text read mode.
  - `delimiter`: Column delimiter. Default: `","`
  - `quotechar`: String delimiter. Default: `None`
- `csv.reader` is iterable using a `for` loop.
  - Variable in each loop is a list of strings representing a row of the table.
  - Read a single line, e.g. the header, using `next(csvReader)`.
- **Documentation:** docs.python.org/3/library/csv

Tables – CSV

```
import csv

with open("data.csv", "w", newline="") as handle:
    writer = csv.writer(handle)
    writer.writerow(["Value", "Squared"])
    for i in range(2, 4):
        writer.writerow([i, i**2])

with open("data.csv", "r") as handle:
    reader = csv.reader(handle)
    header = next(reader)
    print(*header, sep="\t")
    for row in reader:                       # 2    4
        print("{}\t{}".format(*row))          # 3    9
```

`pickle` vs. JSON vs. CSV – When to use what?

- `pickle`
  - Very complex objects. Pure Python projects.
  - No intent to share result with others. Human-readability not required.
- JSON
  - Medium complexity of data objects. Specific order of data not relevant.
  - Intent to share with others. Human-readability required.
- CSV
  - Very simple data types. Table structured data.
  - Exchange data with other programs, in particular spreadsheet programs.

Handling errors – exception handling

- Sometimes it is easier to react to errors than to catch them before they occur.
- Syntax:

```
try:
    statements
except ExceptionClass as variable:
    statements
```

- `variable`: Optional object of type `ExceptionClass` describing the error.
- Statements after `try`: Anything that could go wrong.
- Statements after `except`: Code to handle the error.
- `try` block is left after the first exception has been raised.
- After error treatment program continues normally after the `try` statement.

Exception handling – Exception classes

- Only exceptions specified by exception class in `except` are caught.
- If no exception class is specified all exceptions are caught by this `except`.
- Exceptions have hierarchical order forming groups of exceptions, e.g.,
  - `ZeroDivisionError` is an `ArithmeticError`, but
  - `ArithmeticError` is not necessarily a `ZeroDivisionError`,
  - `ArithmeticError` covers further Exceptions, e.g., `OverflowError`.
- Exception class `Exception` covers (almost) all possible exceptions.
- May define our own exception classes (later).
- Documentation contains a list of pre-defined error classes:
  docs.python.org/3/library/exceptions

Exception handling – Example

```
import math

for x in range(-1,2):
    try:
        print(f"{1/x} {math.sqrt(x)}")
    except Exception as e:
        print(f"Iteration: x={x:+1}, Error: {e}")

    # Iteration: x=-1, Error: math domain error
    # Iteration: x=+0, Error: division by zero
    # 1.0 1.0
```

Handling multiple exceptions

- There may be multiple possible errors in a single `try` block.
- May want to react to some errors in the same way.
  - Replace `ExceptionClass` by a tuple of exception classes.
  - Syntax: `except (ExceptionClass, ExceptionClass) as e`
  - `except` block is run if **any** of the errors is met.
- May want to handle different exceptions differently.
  - May add an arbitrary number of `except` clauses.
  - Only the first matching `except` block is executed.
  - Other `except` clauses - even if matching the error - are ignored.
  - Reminder: Same behavior as `elif` in `if` statements.

Exception handling – Example

```python
import math

for x in range(-1,2):
    try:
        print(f"{1/x} {math.sqrt(x)}")
    except ZeroDivisionError:
        print(f"Division by zero for x={x}")
    except ValueError:
        print(f"Negative value in sqrt for x={x}")

    # Negative value in sqrt for x=-1
    # Division by zero for x=0
    # 1.0 1.0
```

Flux of exception handling

- May attach an `else` block to `try` statements.
  - `else` block is executed if no error occurred.
- May attach a `finally` block to `try` statements.
  - `finally` block is execute in any case.
  - May be used as a safe shutdown for any untreated errors.
- `finally` block is executed after a maybe present `else` block.

Exception handling – Example

```python
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print(f"Division by zero, x={x}, y={y}")
    else:
        print(f"{x} / {y} = {result}")

divide(2,1)      # 2 / 1 = 2.0
divide(2,0)      # Division by zero, x=2, y=0
```

Triggering exceptions

- Sometimes we want to `raise` an exception if something goes or may go wrong.
- Syntax: `raise ExceptionClass(arguments)`
- Arguments: Depends on the particular exception class.
  - Usually a string which will be reported in error handling.

```
confirm = input("Continue? [y/n] ")

if not confirm in ["y", "n"]:
    raise RuntimeError("Invalid choice")
```

Triggering exceptions

- Sometimes we want to pass on exceptions if we can not (fully) treat an error.
- Hopefully a superior functional unit (or end user) treats the problem.
- Syntax: `raise`

```python
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print(f"Division by zero, x={x}, y={y}")
        raise
    else:
        print(f"{x} / {y} = {result}")
```

Recommendations

- Exception classes should be chosen as narrowly as possible.
- Narrow exceptions allow us to treat error more effectively.
- Catching exceptions too unspecifically may lead to wrong treatment.
- Wrong treatment can cause more harm than good.

Types – Classes

- Idea: Data with attached context.
- Classes are data types. Allows us to define our own data types.
- Classes are composed of known data types.

  ⇒ Infinite combination possibilities.
- Class definition: `class ClassName:`
- Instantiation: `classInst = ClassName()`
- Referencing (class) attributes: `ClassName.attribute`
- Referencing (instance) attributes: `classInst.attribute`

Class variables

- Class variables are shared by the class itself and all instantiations.
- Syntax:

```
class ClassName:
    variable = 4
```

- **Note:** Class variables can also be referenced as instance attributes.
- **Warning:** Assignment operations to class variables referenced as instance attribute create a new *instance* variable.

Instance variables

- Instance variables are *unique* to an instance.
- May add new instance variables by assigning values to *non-existing* attributes.
- Syntax:

```
class ClassName:
    classVariable = 4

classInst = ClassName()
classInst.instVariable = 6

# ClassName.instVariable -> AttributeError
```

Example – Immutable class variable

```python
class ClassName:
    variable = 4

classInst = ClassName()

print(ClassName.variable)      # 4
print(classInst.variable)      # 4

classInst.variable = 6
print(ClassName.variable)      # 4
print(classInst.variable)      # 6
```

Example – Mutable class variable

```
class ClassName:
    variable = []

classInst = ClassName()

ClassName.variable.append(1)
classInst.variable.append(2)
print(ClassName.variable)     # [1, 2]
print(classInst.variable)     # [1, 2]

classInst.variable = classInst.variable + [3]
ClassName.variable.append(4)
print(ClassName.variable)     # [1, 2, 4]
print(classInst.variable)     # [1, 2, 3]
```

Class vs. Instance variables

- Class variables: Shared between all instances of a class.
  - **Hint:** Always reference class variables as class attributes.
- Instance variable: Unique to a particular instance.

Methods

- A function that *belongs* to a class.
- Function is defined *within* the class.
- Obligatory first parameter: `self`
  - Reference to the instance on which the method is called.
  - Access instance (and class) attributes via `self`.
  - Passed automatically when calling the method.

    ⇒ One argument less than parameters.
  - Side note: The name `self` is only a convention.
- Methods of a class provide *context* to its variables, i.e., its *data*.

Syntax

```
class ClassName:
    ...
    def method(self, ...):
        ...
    ...

classInst = ClassName()
classInst.method(...)
```

Special methods – Initializer

- Method: `__init__`
- Called when a new instance of a class is initialized.
- Parameters: `self` plus arbitrary others.
- Parameters: Usually used to set default values for instance variables.
- Must return `None`.
- Syntax:

```
class ClassName:
    def __init__(self, ...):
        ...
    ...

classInst = ClassName(...)
```

Example

```
class Coordinates:
  def __init__(self, x = 0.0, y = 0.0, z = 0.0):
    self.x = x
    self.y = y
    self.z = z

treasure = Coordinates(3.4, 2.5, -6.3)
```

Special methods – Text Representation

- Method: `__str__`
  - Called when converting to `str`, i.e., `str(classInst)`.
  - `print` converts all non-string arguments to `str`.
  - No additional parameters except `self`.
  - Must return a string.
- Method: `__repr__`
  - Almost same as `__str__`, but used for debugging, i.e., string contains whatever is required for development.
  - Called by `repr(classInst)`.

Example

```python
class Coordinates:
  def __init__(self, x = 0.0, y = 0.0, z = 0.0):
    self.x = x
    self.y = y
    self.z = z

  def __str__(self):
    return f"(x: {self.x:+}, y: {self.y:+}, z: {self.z:+})"

treasure = Coordinates(3.4, 2.5, -6.3)

print(treasure)     # (x: +3.4, y: +2.5, z: -6.3)
```

Special methods

- Method: `__len__`
  - Called by `len(classInst)`.
  - No additional parameters except `self`.
- Method: `__abs__`
  - Called by `abs(classInst)`.
  - No additional parameters except `self`.
- Methods: `__pos__` and `__neg__`
  - Called by `+classInst` or `-classInst`, respectively.
  - No additional parameters except `self`.
  - Unary arithmetic operators.

Special methods – Function call

- Method: `__call__`
- Called by `classInst(...)`.
- Parameters: `self` plus arbitrary others.
- Allows us to use an instance of a class as if it were a function.

Special methods – Subscription

- Methods: `__getitem__` and `__setitem__`
- Called by `instance[index]`.
- Read access: `__getitem__(self, index)`
  - Parameters: `self` plus `index`.
  - Returns reference to element denoted by `index`.
- Write access: `__setitem__(self, index, value)`
  - Parameters: `self` plus `index` and `value`.
  - Assigns new `value` to element denoted by `index`.
- `index` may be of arbitrary type.
- Multiple arguments in `[]` are packed into a `tuple`.

Special methods – Comparisons

- Methods: `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__lt__`, `__le__`
- Called by comparison operators: `==`, `!=`, `>`, `>=`, `<`, `<=`
- Must return a boolean.
- Binary operators, i.e., two operands.
  - ⇒ Methods must have exactly two parameters.
  - ⇒ `self` is left operand, second parameter, e.g., `rhs`, is right operand.
  - ⇒ Method of the left operand is called.
- **Note:** A class that implements at least `__lt__` or `__gt__` can be used with `sorted` and `listOfInstances.sort()`.

Example

```python
class Coordinates:
  def __init__(self, x = 0.0, y = 0.0, z = 0.0):
    self.x = x
    self.y = y
    self.z = z

  def __eq__(self, rhs):
    return (self.x == rhs.x) and (self.y == rhs.y) \
      and (self.z == rhs.z)

treasure = Coordinates(3.4, 2.5, -6.3)
myPosition = Coordinates(2.4, 1.5, 5.3)

print(treasure == myPosition)     # False
```

Special methods – Binary arithmetic operators

- Methods: `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__truediv__`, `__floordiv__`, `__mod__`, `__pow__`
- Called by arithmetic operators: `+`, `-`, `*`, `@`, `/`, `//`, `%`, `**`
- Binary operators, i.e., two operands.
- **Note:** `@` is usually used for matrix multiplication, hence `__matmul__`, but is also often used for other operations, e.g., cross product.

Left- and right-hand sided binary operators

- So far: For binary operators `self` is the left operand.

  ⇒ The special method of the class on the left-hand side is called.

- Example: `v * 2` calls `__mul__` method of `v`.
- Example: `2 * v` calls `__mul__` method of `int`.
- **Issue**: The required method might not be implemented for the class on the left-hand side or does not support the type on the right-hand side.
- **Solution**: Right-hand sided binary operators.
  - Special methods with prefix `r`, e.g., `__rmul__`.
  - Same as without prefix, but `self` is right operand, the second parameter of the method is the left operand.
  - Special return value `NotImplemented` to indicate unsupported type.

Excursion: Type safety

- Data types are fixed in parameter list for many programming languages.
- Python: Dynamic typing – data types are not fixed.
- May lead to senseless calls, although syntactically correct.
- Manual type safety:

```
class Complex:
    ...
    def __mul__(self, rhs):
        if type(rhs) in (int, float, complex):
            ...
        else:
            return NotImplemented
```

- **Note:** In other cases one might raise an error if types are unsupported.

Special methods – Augmented arithmetic assignments

- In-place variants of the binary operators, i.e., `self` is modified.
- Special methods with prefix `i`, e.g., `__imul__`.
- **Note:** Prefix `i` and `r` for binary operators can not be combined.
- See documentation for further information:
  docs.python.org/3/reference/datamodel

Special methods

- Special methods improve readability, operability and compatibility.
  - E.g., compare `a + b` to `a.add(b)`.
  - Exist for most common operations.
    docs.python.org/3/reference/datamodel
- Define methods in terms of other methods, e.g.

```
class Coordinates:
  def __le__(self, rhs):
    return not self > rhs
```

  - **Note:** `__ne__` is by default implicitly defined as the negation of `__eq__`
    .
- **Hint:** Definitions of special methods shall be consistent with existing ones.

Recommendations

- Common code of two methods shall be factored out into a *shared* method.
  - Both methods call the shared method.
  - Shared methods that are not meant to be explicitly used outside of the class definition are to be *hidden*.
- Attributes of a class that are not meant to be used outside of the class definition are to be *hidden*.
  - Python has no mechanism to *hide* attributes.
  - By convention: *Hidden* attributes are prefixed by a single `_`.
- A class shall have one purpose and one purpose only.

  ⇒ *Single-responsibility principle* (SRP)

Illustration: Classes as professions

- Class instance: *Person with some specific knowledge*
  - *Information* → Instance variables
  - *Knowledge how to interpret existing information* → Methods
- Designing a class: *Describing a profession*
  - *What do people of that profession know?* → Instance and class variables
  - *What questions can they answer?* → Methods
  - *How can they alter their acquired information?* → Methods

Illustration: Classes as professions

- Interaction between classes: *Conversation*
  - One *person* takes the lead and asks questions.
  - *Ask questions* → Call methods.
    - *Whom to ask?* → Instance of another class.
    - *Which question?* → Which method.
    - *How to specify the question?* → Arguments passed to the method.

Example

- *Where, in a given garden, is it a good idea to plant a tree?*
- Two *professions*:
  - Gardener: Knows how to recognize a good spot.
  - Owner of the garden: Knows available spots in their garden.
- Possible approaches:
  - Responsibility with gardener:
    - Asks question: *Which spots are there?*
    - Decides on a good spot based on the reply.
  - Responsibility with the owner:
    - Asks question: *Is a chosen spot a good spot?*
    - Repeats until a positive reply is given.

Whom to give the responsibility?

- Depends on many different factors.
- Some thought that *might* go into consideration:
  - Efficiency: Which approach minimizes the number of function calls?
  - User interface: Which object will be mostly used by the end-user?
  - Hierarchy: Which object already owns most of the required data?
- There might be no definite answer.
- Whichever approach you pick: Stay consistent!

Inheritance

- Idea: Create a new class that is an extension or alteration of an existing class.
- *Base* or *parent* class.
- *Derived* or *child* class
  - Newly created class *derived* from base class.
  - Contains all attributes of its base class.
  - Inherits all attributes from its parent class.
  - May add arbitrary new attributes.
  - May overwrite attributes of base class.
  - Changes affect only the derived class.
- Syntax: `class DerivedClass(BaseClass)`

Inheritance

- Any object of type `DerivedClass` *is* also of type `BaseClass`.
  - ⇒ Methods of base class may be applied on an instance of a derived class.
  - ⇒ Any function that expects arguments of base class shall also work for instances of derived classes.
- May access the base class *part* within the derived class using `super()`.
- Derived classes may be base classes.

Example

```python
class Coordinates:
  def __init__(self, x = 0.0, y = 0.0, z = 0.0):
    self.x = x
    self.y = y
    self.z = z

class Shop(Coordinates):
  def __init__(self, name, x = 0.0, y = 0.0, z = 0.0):
    super().__init__(x, y, z)
    self.name = name

food = Shop("Grocery Store", 5.4, 6.6, 7.8)

print(f"{food.name} ({food.x}, {food.y}, {food.z})")
    # Grocery Store (5.4, 6.6, 7.8)
```

Example

```
import math

def distance(a, b):
  return math.sqrt(
    (a.x - b.x)**2 + (a.y - b.y)**2 + (a.z - b.z)**2)

treasure = Coordinates(3.4, 2.5, -6.3)
myPosition = Coordinates(2.4, 1.5, 5.3)

print(f"{distance(treasure, myPosition):.2f}")     # 11.69

food = Shop("Grocery Store", 5.4, 6.6, 7.8)
drinks = Shop("Liquor Store", 2.2, 4.4, 7.8)

print(f"{distance(food, drinks):.2f}")              #  3.88
```

Multiple Inheritance

- A class may be derived from multiple base classes.
- Syntax: `class DerivedClass(BaseClass1, BaseClass2, ...):`
- **Warning:** May end in name collisions!
  - An attribute name may exist in multiple base classes.
  - Order of base classes in definition matters.
  - How does `super()` work for multiple inheritance?
  - See documentation:
    docs.python.org/3/tutorial/classes#multiple-inheritance
- **Hint:** Avoid using multiple inheritance!

Containment

- Idea: Create a new class re-using functionality of existing classes by *containing* instances thereof.
- The new class does **not** *inherit* any attributes of the contained class(es).
  - Can only indirectly call methods.
  - May add new methods forwarding tasks.
- No further coupling between the classes.
- May combine inheritance and containment.

Example

```
class Shop:
  def __init__(self, name, x = 0.0, y = 0.0, z = 0.0):
    self.name = name
    self.location = Coordinates(x, y, z)

  def __eq__(self, rhs):
    return self.name == rhs.name and \
      self.location == rhs.location

food = Shop("Grocery Store", 5.4, 6.6, 7.8)
drinks = Shop("Liquor Store", 2.2, 4.4, 7.8)

print(f"{distance(food.location, drinks.location):.2f}")
```

When to use inheritance?

- Inheritance leads to high coupling.

  ⇒ The derived class *depends* on the (implementation of the) base class.

- Multiple layers of inheritance increase dependencies and may lead to confusion.
- Multiple inheritance increases dependencies and leads to confusion.
- **Hint:** Avoid overly complicated structures.
- **Hint:** Keep classes self-contained whenever possible.
- **Hint:** In programming in general one strives to minimize dependencies.

Revisited: Exception classes

- In Python exceptions or groups of exceptions are classes.
- Hierarchical structure is achieved through inheritance.
- May add our own exception:
  - Create a class that inherits directly or indirectly from class `Exception`.
  - New exception class may otherwise remain empty.

pass statement

- The `pass` statement does nothing.
- May be used when a statement is only required syntactically.

```
class InputError(Exception):
    pass

while True:
    pass # Busy-wait for keyboard interrupt (Ctrl+C)
```

- May be used as a place-holder when working on new code.

```
def func(*args):
    pass # TODO: Implement function!
```

Generator statement

- Idea: *Lazy evaluated* sequence of values, i.e., values are *generated* on the fly.
- Technically: A function that returns an *iterable* object.
- Syntax: Similar to defining a function, but use `yield` instead of `return`.
- `yield` produces, i.e., *returns*, a value from the generator.
- Values are produced when iterating over the generator.

Example

```python
def myRange(start, stop, step = 1):
    value = start
    while value < stop:
        yield value
        value += step

gen = myRange(1, 9, 2)

print(gen)              # <generator object myRange at 0x...>
print(list(gen))        # [1, 3, 5, 7]
print(list(gen))        # []
```

Generator expression

- Short version to create a generator:

```
gen = (expr for x in iterable if condition)
```

- Example:

```
squares_generator = (i * i for i in range(2,4))

for i in squares_generator:     # 4
    print(i)                    # 9
```

Infinite sequence

- **Note:** Infinite sequences can not be stored in memory.
- Generators allow us to represent infinite sequences.
- Only one item of the sequence exists at a time.
- Example:

```
def count(start = 0, step = 1):
    while True:
        yield start
        start += step

# print(list(count()))

for i in count():
    print(i)
```

Example – Nesting Generators

```
def fibonacci(N):
    x, y = 0, 1
    for _ in range(N):
        yield x
        x, y = y, x+y

def square(nums):
    for num in nums:
        yield num**2

print(sum(square(fibonacci(10))))            # 4895

print(sum((n**2 for n in fibonacci(10))))    # 4895
```

Predefined generators

- `range(stop)`
- `range(start, stop, step = 1)`
- `itertools.count(start = 0, step = 1)`
- `itertools.repeat(object, n)`: Repeat `object` `n` times.
- `itertools.repeat(object)`: Repeat `object` indefinitely.

Functional programming – `map`

- `map` allows us to apply a function to each element in iterables.
- Signature: `map(func, *iterables)`
- **Note:** The function `func` must expect exactly as many arguments as iterables are specified for `*iterables`.
- **Note:** Length of the map is the length of the shortest iterable.
- **Note:** `map` itself is a generator.

Examples

```
names = ["alfred", "william", "klaus"]
print(list(map(str.upper,names)))
    # ['ALFRED', 'WILLIAM', 'KLAUS']

import itertools
values = [3.56773, 5.57668, 4.00914, 56.24241, 9.01344]
print(list(map(round, values, itertools.repeat(2))))
    # [3.57, 5.58, 4.01, 56.24, 9.01]

A = [1, 2, 3]
B = [4, 5, 6]
C = [7, 8, 9]
print(sum(map(lambda a, b, c : a*b+c, A, B, C)))
    # 56
```

Functional programming – `filter`

- `filter` allows us to *filter* an iterable by a given function.
- Signature: `filter(func, iterable)`
- **Note:** The function `func` must expect exactly one argument and return a boolean.
- **Note:** `filter` itself is a generator.

Examples

```
scores = [55, 75, 80, 60]
print(list(filter(lambda score : score > 70, scores)))
    # [75, 80]

words = ("rewire", "madam", "freer", "anutforajaroftuna")
print(list(filter(lambda word : word == word[::-1], words)))
    # ['madam', 'anutforajaroftuna']
```

Functional programming – `functools.reduce`

- `functools.reduce` allows us to *reduce* an iterable to a single value.
- Signature: `functools.reduce(func, iterable, initial = None)`
- **Note:** The function `func` must expect exactly two arguments.
- Example:

```
import functools
import operator

numbers = [3, 4, 6, 9, 34, 12]
print(functools.reduce(operator.add, numbers))        # 68

print(functools.reduce(operator.mul, range(1,5)))    # 24
```

Imports

- Already known: `import module`
  - Grants access to all of the functionality, i.e., *names*, defined in `module`.
  - Accessing names of the imported module requires specifying namespace.
- Selective import: `from module import names`
  - `names`: Comma separated list of names to be imported.
  - Names are *introduced* into current namespace.
  - Imported names are accessed without specifying their original namespace.
  - Use `*` to import all names of a module.
    - This may easily lead to name conflicts.
    - **Hint:** Only import required names.
    - **Warning:** Do not use `*`!

Aliased Imports

- Syntax: `import module as alias`
  - Grants access to all names defined in `module` through `alias`.
- Syntax: `from module import name as alias`
  - Introduce `name` of `module` into the current namespace as `alias`.
  - May specify multiple comma separated names and aliases.
    - **Hint:** Use a single import statement for each name to be aliased.

Examples

- `from math import pi, sin, cos, tan`
- `import foo.bar.baz as fbb`

Split code into multiple files – Why?

- Keeping code in one file may make it easier to write at the beginning.
- With more and more code, reading and modifying it gets more complicated.
- Splitting code into multiple files eases locating pieces of code.
- Facilitates collaboration in teams.
- Makes it easier to track changes under version control.

If each file is concerned with one aspect of your application and every file has a good name, navigating your project will be easy!

In programming, readability and maintainability are more important than writing speed!

Split code into multiple files – How?

- To access code of `myfile.py` in same directory: `import myfile`
  - Must not specify the file extension `.py`.
- When *importing* a file, Python *runs* the file to recognize all defined names.
  - ⇒ Statements of the outermost, i.e., module, scope are executed.

myfile.py

```
print("Hello! I'm a file containing Python code.")
```

run.py

```
import myfile
```

Split code into multiple files – How?

- Importing files works in the same way as importing modules.
- `import myfile`, referring to names as `myfile.name`.
- `from myfile import name`, introducing `name` into current namespace.
- `from myfile import *`, introducing *everything* into current namespace.
- Aliased imports.

How does Python look for an import?

- Python looks in the folder of the executed script for imports before looking for built-in or installed packages (in system paths).
- In interactive mode the interpreter first looks in the current working directory.
- To access files in subfolders specify the relative path to the file using `.` as path separator, e.g., `import subfolder.myfile`.

Don't give your files the same names as (built-in) modules!

- For example, if you create a new file and call it `math.py`, you can not import math of the standard library anymore.

Modes of execution

- When *executing* a script the file is run in *script* mode.
- When importing a file it is run in *module* mode.
- Any file can be run in *script* or *module* mode.
- May even run a file in both modes!
- Variable `__name__` indicates whether a file is run in *script* or *module* mode.
  - In *script* mode `__name__` is set to `"__main__"`.
  - In *module* mode `__name__` is set to the string used to import the file.

⇒ Use `if __name__ == "__main__"` to run code only in *script* mode.

(Regular) Packages

- We may define our own packages to allow easily importing multiple files.
- May be imported by its name.
- Implemented as a directory containing a `__init__.py` file.
- Package is named by the directory name.
- When importing a package `__init__.py` is run in *module* mode.
- May contain subdirectories, which are themselves packages, i.e., subpackages.
- To import subpackages specify all higher-level package(s) using `.` as separator.
- When importing a subpackage the `__init__.py` of the higher-level package(s) and its own are run in *module* mode.

Relative Imports

- Within a package *hierarchy* we may use *relative* imports.
- Syntax: `from .package import ...`
- The leading `.` indicates relative to the current directory.
- Each adding `.` to the leading `.` means one directory up.
- Relative imports only work with the `from ...`, not `import ...` syntax.

Run a package/module in *script* mode

- For packages: `python <package>`
- For modules: `python -m <module>`
- Requirements: `__main__.py` in top directory of package/module.

What else to know about modules and packages?

- Documentation: docs.python.org/3/reference/import
- Tutorial: docs.python.org/3/tutorial/modules
- For now: Treat *package* and *module* as synonyms.

Installing third-party packages

- Depends on how a package is set up.
- Some might require manual installation.
- Most are available using a *package manager*, e.g., *pip* or *conda*.
- pip (an acronym of ”pip Install Packages”) is the *official* standard tool.

  ⇒ Only cover pip.
- Will only cover pip
- pip may use different *repositories* as sources.
- Default repository: Python Package Index (PyPI)

Installing third-party packages – pip

- Ensure pip is installed by running (in a terminal):

  ```
  python -m pip --version
  ```

  – If not, install pip by running:

    ```
    python -m ensurepip --default-pip
    ```

- Ensure `pip`, `setuptools` and `wheels` are up to date by running:

  ```
  python -m pip install --upgrade pip setuptools wheel
  ```

Installing third-party packages – pip

- Install a package from PyPI:

  ```
  python -m pip install package
  ```

- Upgrade an already installed package:

  ```
  python -m pip install --upgrade <package>
  ```

- Install a specific version:

  ```
  python -m pip install "package==version"
  ```

- Install a version greater equal and less than specified versions:

  ```
  python -m pip install "package>=1,<2"
  ```

- Install a version *compatible* with a certain version:

  ```
  python -m pip install "package~=1.4.2"
  ```

Installing third-party packages – pip

- **Note:** In case of multiple installed Python interpreters packages are always installed for the interpreter used to run pip.

- Uninstall a package:

  ```
  python -m pip uninstall <package>
  ```

- List installed packages:

  ```
  python -m pip list
  ```

- Show information about an installed package:

  ```
  python -m pip show <package>
  ```

- Further pip commands and options:

  ```
  python -m pip --help
  ```

System and user site-packages

- Python distinguishes between packages installed system-wide or for a user.
- Packages in user site overwrites system-wide packages.
- pip by default installs packages system-wide.
- Use `pip install --user` to install packages for the current user only.
- Where are *user* packages installed to?

```
python -m site --user-base --user-site
```

Why install to user site?

- No permissions to install system-wide packages.
- Users might require different versions of packages.
- Packages may conflict with other packages.

How to handle projects with conflicting dependencies?

Virtual Environments

- *Virtual Environments* allow packages to be installed in an isolated location for a particular application instead of system-wide or in user site.
- Create a virtual environment: `python -m venv <name>`
  - Creates a new directory named `name` in the current working directory.
- *Activate* a virtual environment:

  Unix: `source <path-to-venv>/bin/activate`

  Windows: `<path-to-venv>\Scripts\activate`

  - Sets up your *environment* to run `python` within the virtual environment.
- *Deactivate* currently active virtual environment: `deactivate`

- By default virtual environments have no access to system-wide or user site installed packages.
- Create a virtual environment with access to system-wide and user site installed packages: `python -m venv --system-site-packages <venv>`
- Further venv commands and options:

  ```
  python -m venv --help
  ```

- **Note:** In case of multiple installed Python interpreters virtual environments always use the same interpreter used to create it.
- **Note:** Running pip inside a virtual environment the `--user` option has no effect. All installation commands will affect the virtual environment.

Installing project requirements

- Transferring a python project to another user or computer one wants to ensure all dependencies, its *requirements*, are installed.
- Usually done using virtual environments.
- Get current list of installed packages in requirements format:

  ```
  python -m pip freeze > requirements.txt
  ```

- Install all dependencies listed in `requirements.txt`

  ```
  python -m pip install -r requirements.txt
  ```

- **Hint:** Add a `requirements.txt` to your project.

Python Notebooks

- A web-based interactive computing platform.
- Third-party package(s) by Jupyter.
  - Next-generation: JupyterLab
    - Install: `python -m pip install jupyterlab`
    - Run: `jupyter lab`
  - Previous-generation: Jupyter Notebook
    - Install: `python -m pip install notebook`
    - Run: `jupyter notebook`

Python Notebooks

- Support different *kernels*.
- Install currently active virtual environment as a kernel:

```
python -m pip install ipykernel
python -m ipykernel install --user \
      --name <name> --display-name "..."
```

- Run system commands inside a notebook: `!{sys.executable} <command>`

  Example: `!{sys,executable} -m pip install matplotlib`

JupyterHub

- Multi-user server for JupyterLab and Jupyter Notebooks.
- Spawns, manages, and proxys single-user server(s).

phyhub.ur.de

Matplotlib: Visualization with Python

- Create high quality plots.
- Create interactive figures.
- Customize visual style and layout.
- Supports various different backends.
- May be embedded in JupyterLab.
- Many third-party packages depend on Matplotlib.
- `python -m pip install matplotlib`

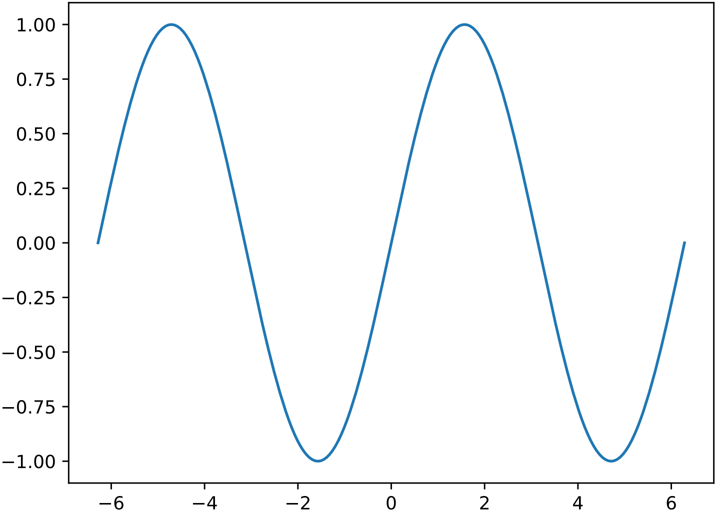
matplotlib.pyplot

- State-based interface to matplotlib.
- Provides an implicit, MATLAB-like, way of plotting.
- Mainly intended for interactive plots.
- Limited to simple cases of programmatic plot generation.
- By convention: `import matplotlib.pyplot as plt`

Example

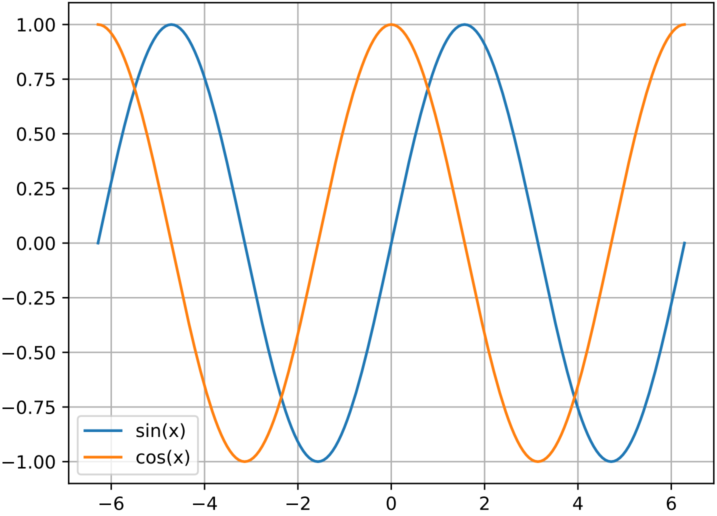
```
import math
import matplotlib.pyplot as plt

N = 100
X = [2 * math.pi * x / N for x in range(-N, N + 1)]
Y = [math.sin(x) for x in X]

plt.plot(X, Y)
plt.show()
```

- `X` and `Y`: Lists of same length setting x and y coordinates of data points.
- `plt.plot(X, Y)`: Prepare plot in memory. May be used to alter settings.
- `plt.show()`: Show the plot. Execution is halted until plot window is closed.


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−6
−4
−2
0
2
4
6

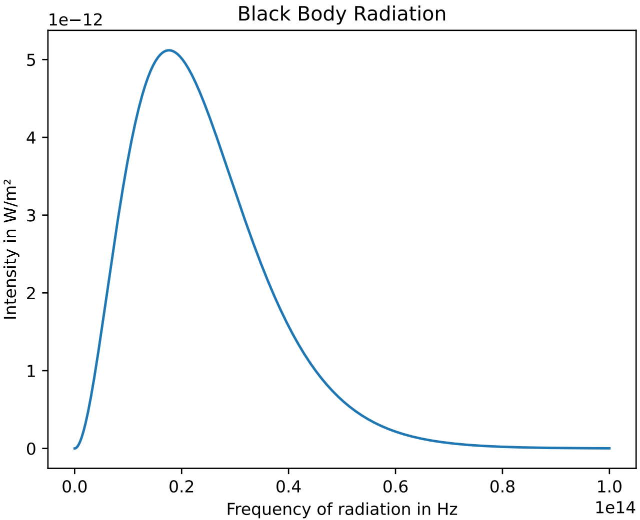
Example – Legend and grid

```python
import math
import matplotlib.pyplot as plt

N = 100
X = [2 * math.pi * x / N for x in range(-N, N + 1)]
Y1 = [math.sin(x) for x in X]
Y2 = [math.cos(x) for x in X]

plt.plot(X, Y1, label = "sin(x)")
plt.plot(X, Y2, label = "cos(x)")

plt.legend()
plt.grid()
plt.show()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−6
−4
−2
0
2
4
6
sin(x)
cos(x)

Example – Title and axis labels

```
h = 6.62607015e-34     # Planck constant
T = 300                # temperature in Kelvin
c = 299792458          # speed of light
kB = 1.380649e-23      # Boltzmann constant

spectralDensity = lambda nu : ((2 * h * nu**3) / (c**2)) / \
    (math.exp((h * nu) / (kB * T)) - 1)

X = [x for x in range(1, int(1e+14), int(1e+10))]
Y = [spectralDensity(x) for x in X]

plt.title ("Black Body Radiation")
plt.xlabel("Frequency of radiation in Hz")
plt.ylabel("Intensity in W/šm")

plt.plot(X, Y)
plt.show()
```

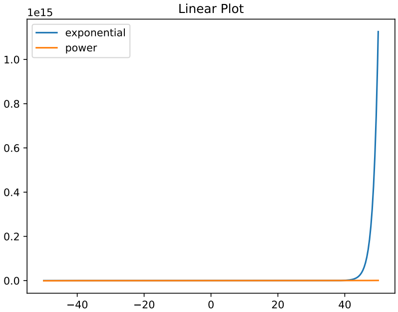

Black Body Radiation
1e−12
5
4
3
2
1
0
Intensity in W/m²
0.0
0.2
0.4
0.6
0.8
1.0
1e14
Frequency of radiation in Hz

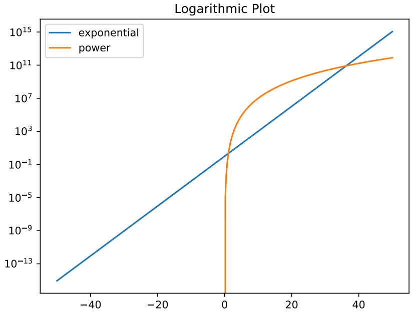
Remarks

- `plt.plot`
  - Each call to `plt.plot` adds a new line to the plot.
  - Automatically picks a color if not explicitly specified.
  - After `plt.show()` all lines are removed.
  - Optional parameter `label`: Sets label of dataset to be added to legend.
- `plt.legend()`: Adds a legend to the plot.
- `plt.gird()`: Shows a grid.
- `plt.title`: Sets the plot title.
- `plt.xlabel` and `plt.ylabel`: Sets the label of the x or y axis.

Example – Logarithmic Plot

```
N = 250
X = [50 * x / N for x in range(-N, N + 1)]
Y1 = [2 ** x for x in X]
Y2 = [x ** 7 for x in X]

plt.title("Linear Plot")
plt.plot(X, Y1, label = "exponential")
plt.plot(X, Y2, label = "power")
plt.legend()
plt.show()

plt.title("Logarithmic Plot")
plt.yscale("log")
plt.plot(X, Y1, label = "exponential")
plt.plot(X, Y2, label = "power")
plt.legend()
plt.show()
```





- Change one or both axes to logarithmic scale.

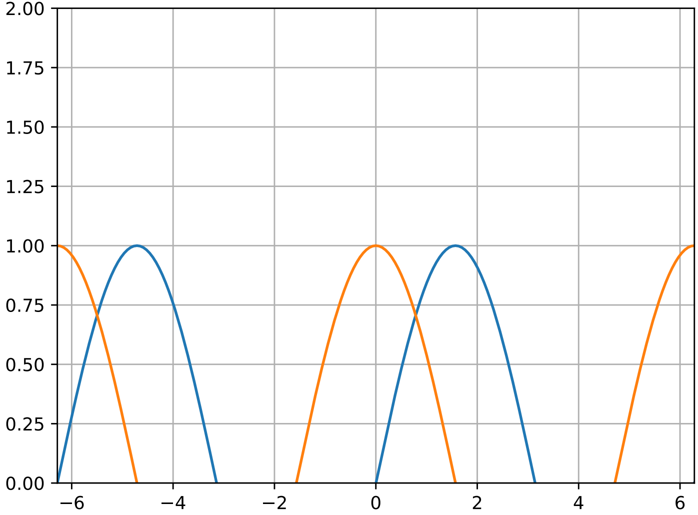
  ⇒ Replace `plt.plot(X, Y)` by `plt.loglog(X, Y)` to scale both axes.

- Does not work for non-positive values.

  ⇒ Use keyword argument `nonpositive` to change default behavior.

Example – Manual limits

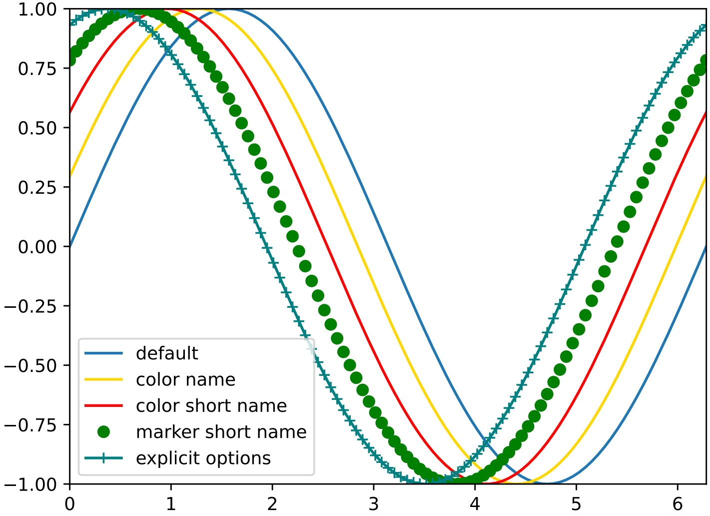
```
N = 100
X = [2 * math.pi * x / N for x in range(-N, N + 1)]
Y1 = [math.sin(x) for x in X]
Y2 = [math.cos(x) for x in X]

plt.plot(X, Y1, label = "sin(x)")
plt.plot(X, Y2, label = "cos(x)")

plt.xlim(-2 * math.pi, 2 * math.pi)
plt.ylim(0, 2)
plt.grid()
plt.show()
```


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−6
−4
−2
0
2
4
6

Example – Line style and markers

```python
N = 100
X = [2 * math.pi * x / N for x in range(-N, N + 1)]

offset = 0
Y = [math.sin(x + offset) for x in X]
plt.plot(X, Y, label = "default")

offset += 0.3
Y = [math.sin(x + offset) for x in X]
plt.plot(X, Y, "gold", label = "color name")

offset += 0.3
Y = [math.sin(x + offset) for x in X]
plt.plot(X, Y, "r", label = "color short name")
```

```python
offset += 0.3
Y = [math.sin(x + offset) for x in X]
plt.plot(X, Y, "go", label = "marker short name")

offset += 0.3
Y = [math.sin(x + offset) for x in X]
plt.plot(X, Y, color = "#008080", linestyle = "-", \
    marker = "+", label = "explicit options")

plt.legend()
plt.xlim(0, 2 * math.pi)
plt.ylim(-1, 1)

plt.show()
```

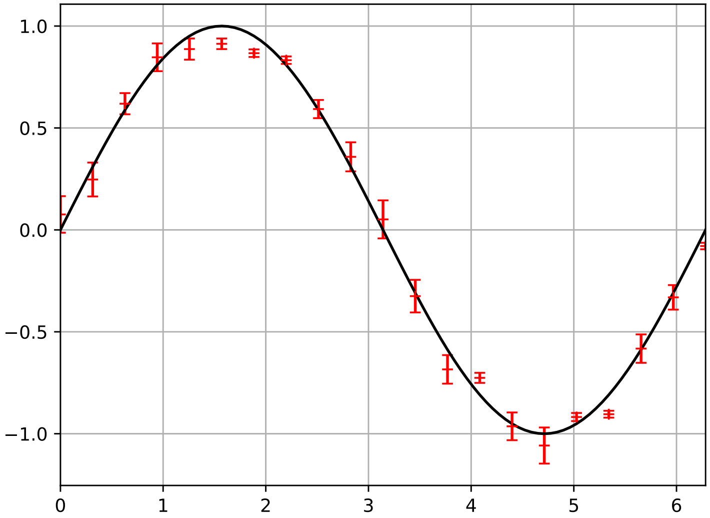

1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5
6
default
color name
color short name
marker short name
explicit options

Manual styling

- Keyword parameters (with default arguments) of the `plot` function to control:
  - Line color and style.
  - Marker color and style.
  - ...
- Strings for most common styles.
- References:
  - Linestyles
  - Markers
  - Colors

Example – Plot with error bars

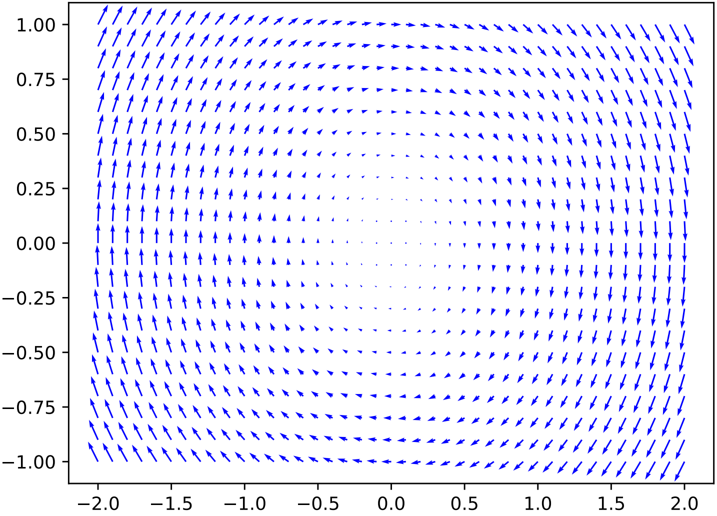
```
N = 20
X = [2 * math.pi * x / N for x in range(0, N + 1)]
Y = [math.sin(x) + random.uniform(-1e-1, 1e-1) for x in X]
err = [random.uniform(1e-2, 1e-1) for _ in X]
plt.errorbar(X, Y, yerr = err, capsize = 3, fmt = 'r+')

N = 100
RX = [2 * math.pi * x / N for x in range(0, N + 1)]
RY = [math.sin(x) for x in RX]
plt.plot(RX, RY, 'k')

plt.xlim(0, 2 * math.pi)
plt.grid()
plt.show()
```



1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5
6

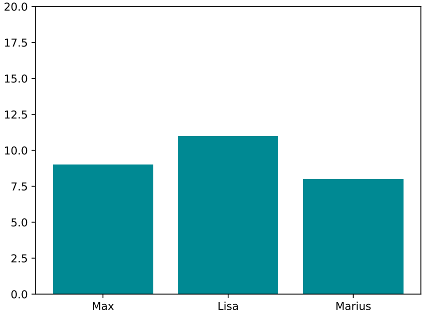
Example – Quiver

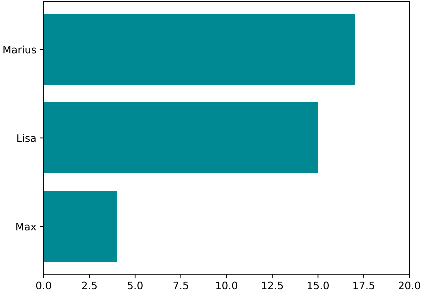
```
W, H = 20, 10
xPoints = [2 * x / W for x in range(-W, W + 1)]
yPoints = [1 * y / H for y in range(-H, H + 1)]

nX, nY = len(xPoints), len(yPoints)
X = xPoints * nY
Y = [yPoints[i] for i in range(nY) for _ in range(nX)]

U = Y.copy(); V = X.copy()     # vector field: v = (y, -x)
for i, v in enumerate(V):
    V[i] = -v

plt.quiver(X, Y, U, V, 'b')
plt.show()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

Example – Bar plots

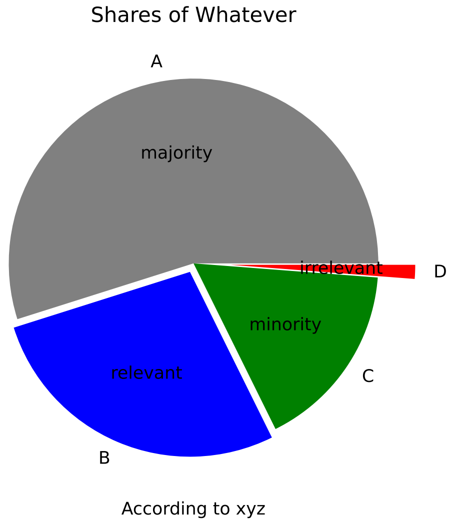
```
import matplotlib.pyplot as plt
import random

X = ["Max", "Lisa", "Marius"]
Y = [random.randint(4, 20) for _ in X]
plt.bar(X, Y, color = '#008993')
plt.ylim(0, 20)
plt.show()

X = ["Max", "Lisa", "Marius"]
Y = [random.randint(4, 20) for _ in X]
plt.barh(X, Z, color = '#008993')
plt.xlim(0, 20)
plt.show()
```


20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Max
Lisa
Marius


Marius
Lisa
Max
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

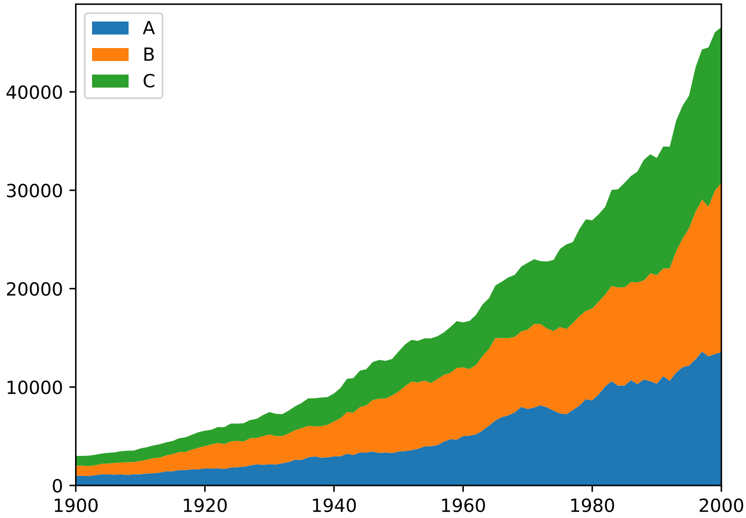
Example – Pie plot

```
def assessPercentage(x):
    if x < 5: return "irrelevant"
    elif 5 < x < 20: return "minority"
    elif 20 < x < 50: return "relevant"
    else: return "majority"

shares = {"A": 90, "B": 45, "C": 27, "D": 2}
plt.title("Shares of Whatever")
plt.xlabel("According to xyz")

plt.pie(shares.values(), labels = shares.keys(), \
    autopct = assessPercentage, explode = (0, .05, 0, .2), \
    colors = ["grey", "b", "g", "r"])
plt.show()
```


Shares of Whatever
A
majority
irrelevant
D
minority
C
relevant
B
According to xyz

Example – Stack plot

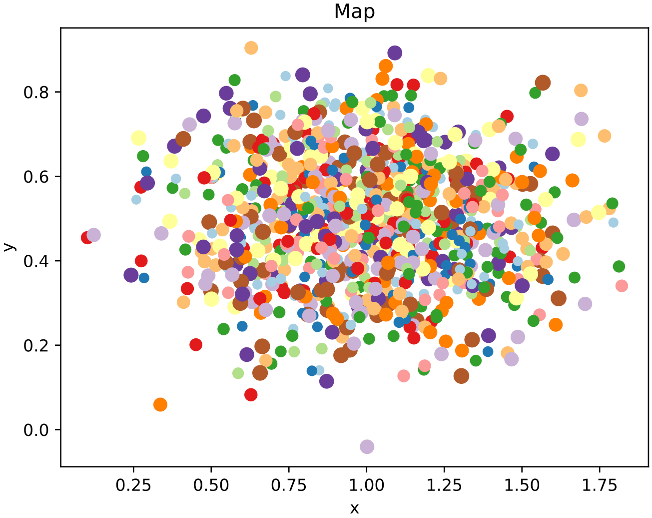
```python
import matplotlib.pyplot as plt
import random

years = list(range(1900,2001))
A, B, C = [1000], [1000], [1000]
for _ in range(1, len(years)):
    A.append(A[-1] * random.uniform(0.95, 1.1))
    B.append(B[-1] * random.uniform(0.95, 1.1))
    C.append(C[-1] * random.uniform(0.95, 1.1))

plt.stackplot(years, A, B, C, labels = ["A", "B", "C"])
plt.xlim(1900,2000)
plt.legend(loc = "upper left")
plt.show()
```



A
B
C
0
10000
20000
30000
40000
1900
1920
1940
1960
1980
2000

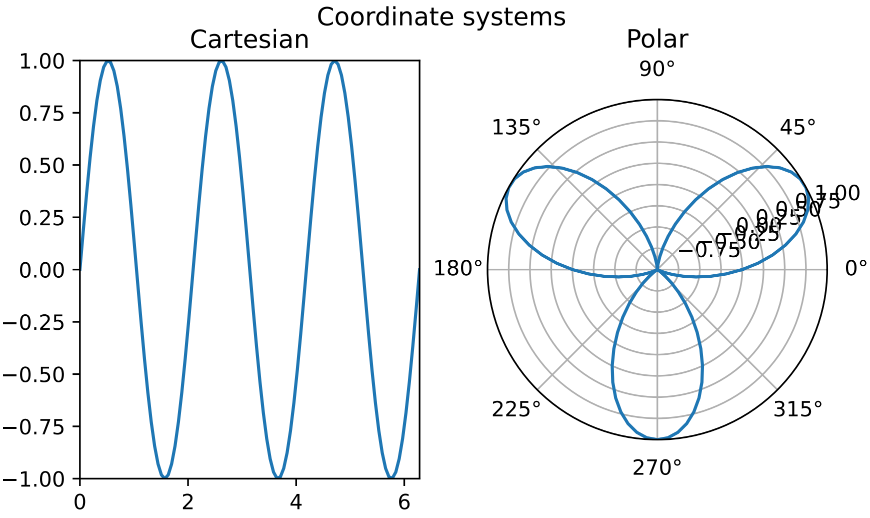
Example – Scatter plot

```
x, y, size, color = [], [], [], []
for _ in range(1000):
    x.append(random.gauss(1, 0.3))
    y.append(random.gauss(0.5, 0.15))
    size.append(random.uniform(25, 75))
    color.append( \
        matplotlib.colormaps['Paired']((size[-1]-25)/50))

plt.title("Map"); plt.xlabel("x"); plt.ylabel("y");

plt.scatter(x, y, size, color)
#plt.scatter(x, y, size, c = Size, cmap = 'Paired')

plt.show()
```


Map
0.8
0.6
0.4
0.2
0.0
y
0.25
0.50
0.75
1.00
1.25
1.50
1.75
x

Example – Figure and subplot

```
N = 100
X = [2 * math.pi * x / N for x in range(0, N + 1)]
Y = [math.sin(3 * x) for x in X]

plt.figure(figsize = (6.4, 3.6))
plt.suptitle("Coordinate systems")

plt.subplot(121)
plt.title("Cartesian")
plt.margins(0, 0)
plt.plot(X, Y)

plt.subplot(1, 2, 2, polar = True)
plt.title("Polar")
plt.margins(0, 0)
plt.plot(X, Y)

plt.show()
```


Coordinate systems
Cartesian
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
Polar
90°
45°
135°
180°
0°
225°
315°
270°

- `plt.suplot` defines a grid and sets selected plot to be active.
- Parameters: `rows`, `columns`, `index`
- Index starts at 1 in the upper left corner and increases to the right.
- Subsequent commands affect activate plot only.
- Optional parameters:
  - `polar`: Plots in polar coordinates.
  - `sharex` and `sharey`: Share axes with other plots.

Backends

- Matplotlib supports multiple *interactive* and *static* backends.
  - Interactive backends: `matplotlib.rcsetup.interactive_bk`
  - Static backends: `matplotlib.rcsetup.non_interactive_bk`
  - All backends: `matplotlib.rcsetup.all_backends`
- (Usually) not required to set backend manually.
- Default backend depends on the environment.

Saving plots to files

- Replace `plt.show()` by `plt.savefig(filename)`.
- Optional parameter `format`: Sets the file format.
  By default deduced by the extension of the filename.
- Optional parameter `backend`: Sets the backend to be used.
  By default auto-detected depending on the set format.
- Further optional parameters:
  - Paper type and orientation
  - Transparency, colors and padding
  - Resolution in dots per inch
  - Metadata
- Reference

Example – Save to pdf

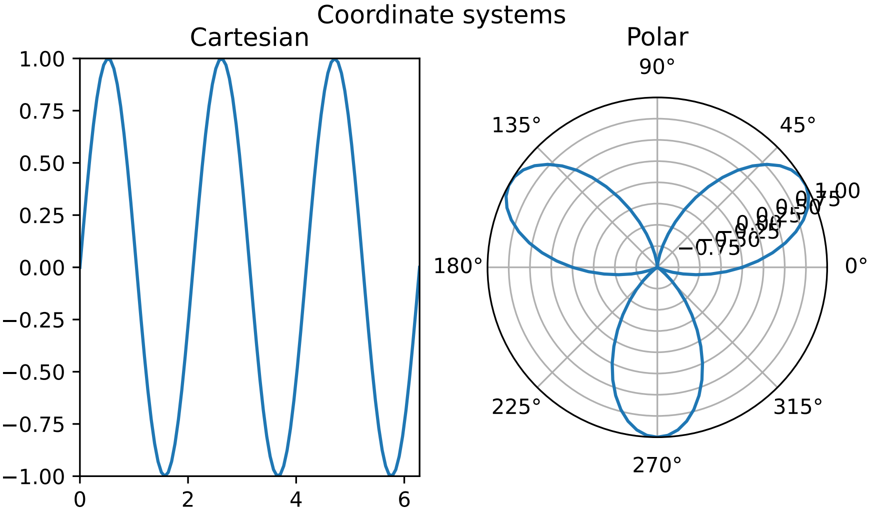
```
import math
import matplotlib.pyplot as plt

N = 100
X = [2 * math.pi * x / N for x in range(-N, N + 1)]
Y = [math.sin(x) for x in X]

plt.plot(X, Y)

plt.savefig("sin.pdf")
```

Object-oriented approach to Matplotlib

- Idea: Plots and elements thereof are objects.
- Allows more specific composition of plots.
- Allows dynamic evolution of plots, e.g., on trigger.
- Primary objects: `Figure` and `AxesSubplot`
  - `Figure`: Window containing plots
  - `AxesSubplot`: Region in the window where a plot can be placed.
- `show()` behaves differently:
  - `plt.show()`: Shows all figures at once, halts execution.
  - `fig.show()`: Shows only the figure `fig`, does not halt execution.
- `fig.savefig()` to save a figure.

Example – Object-oriented approach

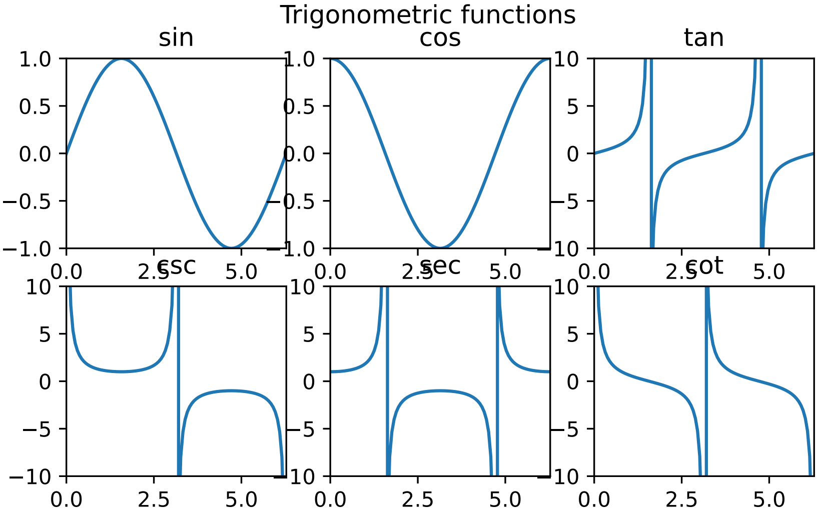
```python
import matplotlib.pyplot as plt

fig = plt.figure(figsize = (6.4, 3.6))
fig.suptitle("Coordinate systems")

crt = fig.add_subplot(1, 2, 1)
crt.set_title("Cartesian")
crt.plot(X, Y)

pol = fig.add_subplot(1, 2, 2, projection="polar")
pol.set_title("Polar")
pol.plot(X, Y)

fig.show()
```



Coordinate systems
Cartesian
Polar
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
90°
45°
135°
180°
0°
225°
270°
315°

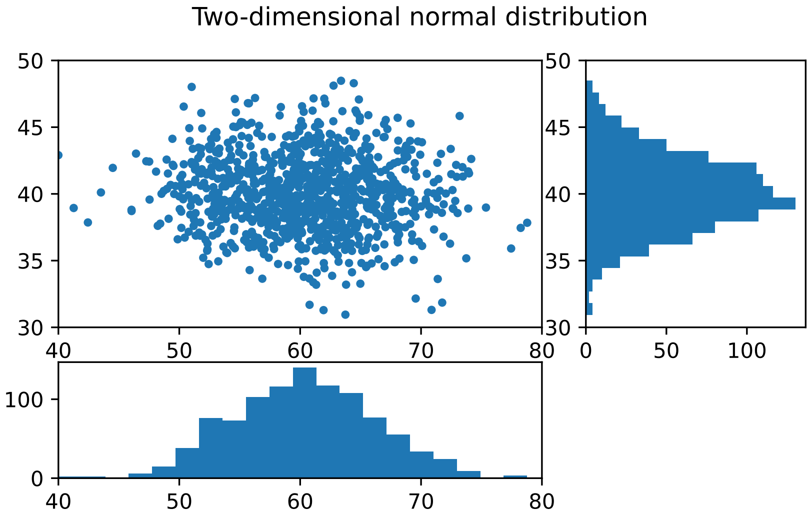
Example – Subplots

```
fig, ((sin, cos, tan), (csc, sec, cot)) = \
    plt.subplots(3, 2, figsize = (6.4, 3.6))
fig.suptitle("Trigonometric functions")

sin.set_title("sin")
sin.plot(X, SIN)

cos.set_title("cos")
cos.plot(X, COS)

...

fig.show()
```


Trigonometric functions
sin
cos
tan
csc
sec
cot
1.0
0.5
0.0
−0.5
−1.0
10
5
0
−5
−10
0.0
2.5
5.0

Example – Gridspec

```
X = [random.gauss(60, 6) for _ in range(1000)]
Y = [random.gauss(40, 3) for _ in range(1000)]

fig = plt.figure(figsize = (6.4, 3.6))
fig.suptitle("Two-dimensional normal distribution")

gs = fig.add_gridspec(3,3)
fig.subplots_adjust(wspace = 0.2, hspace = 0.3)

scatter = fig.add_subplot(gs[0:2, 0:2])
scatter.set_xlim(40, 80)
scatter.set_ylim(30, 50)
histX = fig.add_subplot(gs[2, 0:2], sharex = scatter)
histY = fig.add_subplot(gs[0:2, 2], sharey = scatter)

scatter.scatter(X, Y, marker = ".")
histX.hist(X, orientation = "vertical", bins = 20)
histY.hist(Y, orientation = "horizontal", bins = 20)
```


Two-dimensional normal distribution
50
45
40
35
30
40
50
60
70
80
0
50
100
0
100

Ticks

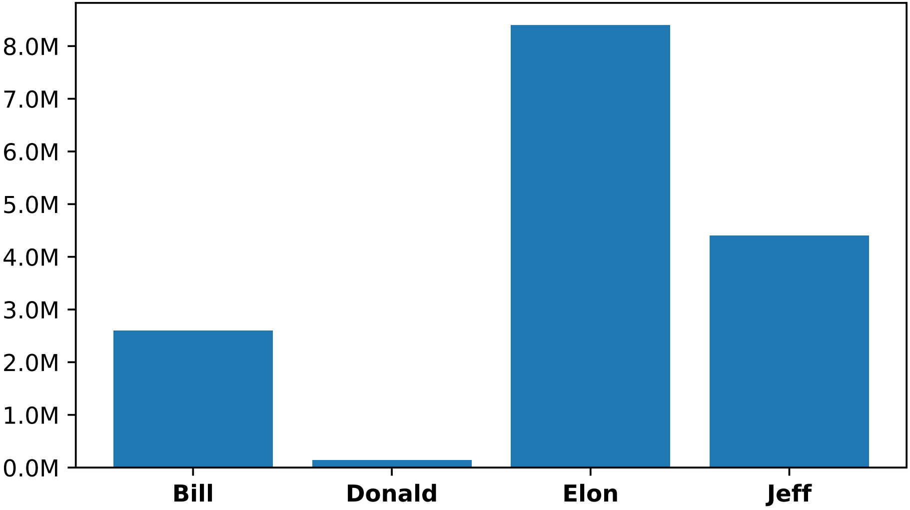
- State-based: `plt.xticks(ticks, lablels, minor)`
- Object-oriented: `axes.set_xticks(ticks, labels, minor)`
- `ticks`: Iterable object specifying where to place tick marks.
- `labels`: Optional iterable argument to specify text to place at tick marks.
- `minor`: Whether the ticks are minor ticks or not, i.e., major ticks (default).
- Optional argument `fontdict`: Specify Text format for labels.
- `axes.set_xticklabels(labels, minor)`: Method to only set labels.
- Same for y-axis, replacing `x` by `y`.

Ticker Funcformatter

- Generate tick labels according to a function.
  - Parameters: `value` and `position`
  - Returns: `str`
- Must be wrapped into Matplotlib interface
- Dedicated class `matplotlib.ticker.FuncFormatter`.

Example – Ticks

```python
from matplotlib.ticker import FunFormatter
import matplotlib.pyplot as plt

P = ["Bill", "Donald", "Elon", "Jeff"]
M = [2.6e6, 1.4e5, 8.4e6, 4.4e6]

fig, ax = plt.subplots(figsize = (6.4, 3.6))
ax.bar(range(len(M)), M)
ax.set_xticks(range(len(M)))
ax.set_xticklabels(P, fontdict = {"fontweight": "bold"})

ylabels = lambda x, pos : f"{x*1e-6:1.1f}M"
formatter = FuncFormatter(ylabels)
ax.yaxis.set_major_formatter(formatter)
```

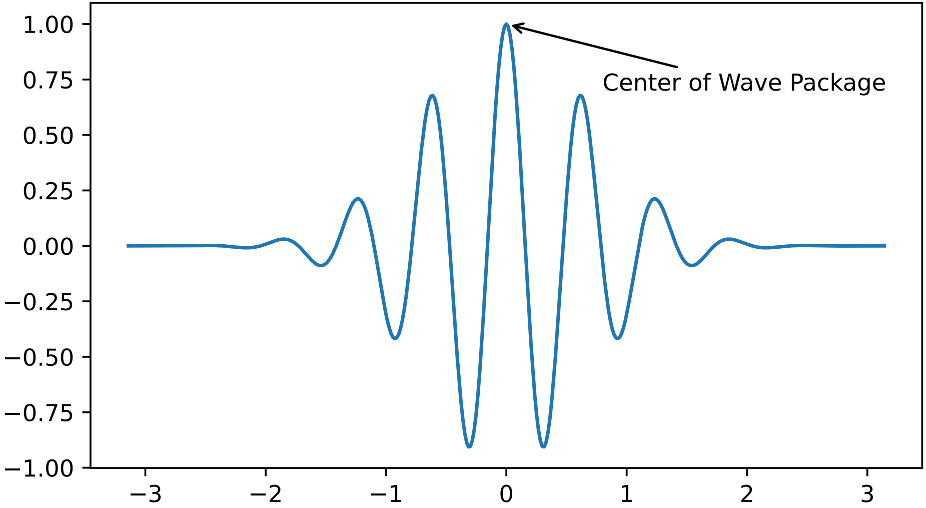

8.0M
7.0M
6.0M
5.0M
4.0M
3.0M
2.0M
1.0M
0.0M
Bill
Donald
Elon
Jeff

Text overlays

- Freely place additional Text in figure.
- State-based: `plt.annotate(...)`
- Object-oriented: `axes.annotate(...)`
- Mandatory arguments:
  - `text`: What to print on figure.
  - `xy`: Coordinates as a `tuple`.
- Many more optional arguments, see Reference.

Example – Annotations with arrows

```
import math
import matplotlib.pyplot as plt

N = 200
X = [math.pi * x / N for x in range(-N, N + 1)]
Y = [math.cos(10*x) * math.exp(-x**2) for x in X]

fig, ax = plt.subplots(figsize = (6.4, 3.6))

ax.plot(X, Y)
ax.annotate("Center of Wave Package", \
    xy = (0, 1), xytext = (0.8, 0.7), \
    arrowprops={'arrowstyle' : '->'})
```



Center of Wave Package
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−3
−2
−1
0
1
2
3

Dynamically modifying plot elements

- Most methods actual return objects, e.g., `plot`, `pie`.
- These objects represent plot elements.
- May be used to change elements dynamically.
- Reference
- Updated in memory only.
- Call `fig.canvas.draw()` to update on screen.

Example

```
import matplotlib.pyplot as plt

N = 100
X = [2 * math.pi * x / N for x in range(-N, N + 1)]
Y = [math.sin(x) for x in X]

fig, ax = plt.subplots(figsize = (6.4, 3.6))

p = ax.plot(X, Y)
fig.show()

p.set_color("g")
fig.canvas.draw()
```

Three-dimensional curve plots

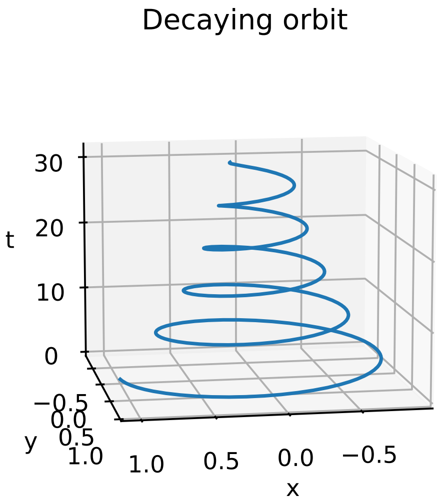
- Specialized Matplotlib submodule adds support for three-dimensional plots.
- `from mpl_toolkits.mplot3d import Axes3D`
- Set projection to `"3d"`, e.g.,

  ```
  ax = fig.add_subplot(projection = "3d")
  ```
- `plot(...)` then supports arguments `X`, `Y` and `Z`.

  ⇒ Three-dimensional curves

Example – Three-dimensional curve

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

T = [math.pi * t / 100 for t in range(1000)]
X = [math.exp(-0.05 * t) * math.cos(t) for t in T]
Y = [math.exp(-0.05 * t) * math.sin(t) for t in T]

fig, ax = plt.subplots(figsize = (6.4, 3.6), \
    subplot_kw={"projection": "3d"})
ax.set_title("Decaying orbit")
ax.set_xlabel("x"); ax.set_ylabel("y"); ax.set_zlabel("t");

ax.view_init(10, 80)
ax.plot(X, Y, T)
```


Decaying orbit
30
20
10
0
t
−0.5
0.0
0.5
1.0
y
1.0
0.5
0.0
−0.5
x

Three-dimensional surface plots

Very complex to setup without help of another very useful module...

NumPy: Scientific computing with Python

- Fundamental package used for scientific computing.
- Features:
  - Mathematical functions.
  - Linear algebra.
  - Random number generators.
  - Fourier transforms.
- Performant!

  The core of NumPy is well-optimized C code.
- By convention: `import numpy as np`

Numpy: Data types

- Numpy defines its own data types mapping to C.
- Are not as flexible as other data types we have seen so far in Pyton.

  ⇒ Reduction of flexibility for performance.
- Boolean: `np.bool_`
- Integer: `np.int8`, `np.int16`, `np.int32`, `np.int64`
- Unsigned: `np.uint8`, `np.uint16`, `np.uint32`, `np.uint64`
- Float: `np.float16`, `np.float32`, `np.float64`
  `np.half`, `np.single`, `np.double`
- Complex: `np.complex64`, `np.complex128` `np.csingle`, `np.cdouble`
- (Custom) Structured Data Types

Array objects

- Index N-dimensional array type `np.ndarray`.
- Collection of objects of same data type.
- Use `np.array` function to convert Python iterables to `np.ndarray`.
  - Implicit type conversions.
  - Works with nested lists if homogenous.
  - Example:

```
a = np.array([1, 2, 3])       # [1 2 3]
b = np.array([1.0, 2.0])      # [1. 2.]
c = np.array([1.0, 2])        # [1. 2.]

d = np.array([[1, 2, 3], [4, 5, 6]])      # [[1 2 3]
                                          #  [4 5 6]]
```

Selected attributes of np.ndarray

- `dtype`: Data type of all element in collection .
  - Use optional parameter `dtype` of `np.array` to force data type.
- `shape`: Tuple of elements per dimension.
- `size`: Total number of elements.
  - `array.size == prod(array.shape)`
- `ndim`: Number of dimensions.
  - `array.ndim == len(array.shape)`
- `data`: Memory address of data.
  Used for advanced techniques and interoperability with external libraries.

Accessing elements of np.ndarray

- Same notation as for lists: Indices and slices.
- Supports specifying multiple indices or slices in `[ ]`, one for each dimension.
  - `array[i, j]` is equivalent to `array[i][j]`
  - `array[:, j]` means *all* rows of column `j`
  - `array[i:j, k:l]` is a submatrix of rows `i` to `j` and columns `k` to `l`.
- List or `np.ndarray` of indices for each dimension.
  - Have to be of same length for all dimensions.
  - Returns *flattened* `np.ndarray` if used with multiple indices.
- `np.ndarray` of booleans of same shape to specify which elements to select.
  - Returns *flattened* `np.ndarray` if used with multiple indices.

Examples

```
array = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

print(array[0, 1])          # 2
print(array[-1])            # [7 8 9]
print(array[:, 1])          # [2 5 8]
print(array[::2, ::2])      # [[1 3]
                            #  [7 9]]

print(array[[0, 1], [1, 2]]) # [2 6]
print(array[[[True, False, True],
             [False, True, False],
             [True, False, True]]] # [1 3 5 7 9]
```

Commonly used array objects

- `np.empty(shape)`: `np.ndarray` with uninitialized entries.
- `np.zeros(shape)`: `np.ndarray` filled with zeros.
- `np.ones(shape)`: `np.ndarray` filled with ones.
- `np.full(shape, value)`: `np.ndarray` filled with `value`.
- `np.identity(n)`: Identity matrix I_n.
- `np.diag(obj)`: Extract a diagonal or construct a diagonal array.
  - If `obj` is a 1-D array: Returns a 2-D array with `obj` on its diagonal.
  - If `obj` is a 2-D array: Returns the diagonal of `obj` as a 1-D array.
- **Note:** `shape` may be a single dimension or a tuple of dimensions.
- **Note:** Use optional parameter `dtype` to set data type of the created array.

Examples

```
print(np.zeros(5))                            # [0.  0.  0.  0.  0.]
print(np.zeros(5, dtype = np.int32))          # [0 0 0 0 0]

print(np.empty(5))                            # [? ? ? ? ?]

print(np.ones((2,2)))                         # [[1.  1.]
                                              #  [1.  1.]]

print(np.full((2,2), math.pi))                # [[3.141 3.141]
                                              #  [3.141 3.141]]

print(np.identity(2))                         # [[1.  0.]
                                              #  [0.  1.]]
print(np.diag(np.identity(2)))                # [1.  1.]
```

Auxiliary array objects

- `np.arange(start, stop, stride)`:
  - Returns evenly spaced values within given **half-open** interval.
  - For integers it is roughly equivalent to the Python built-in `range`.
  - Not recommended to be used with floating-point values.
- `np.linspace(start, stop, N = 50)`:
  - Returns `N` evenly spaced values within given **closed** interval.
  - Example: `np.linspace(2, 8, 3) == np.array([2, 5, 8])`
- `np.logspace(start, stop, N = 50, base = 10)`:
  - Returns `N` evenly spaced values on log scale with base `base` within given closed interval from `base ** start` to `base ** stop`.
- And many more array creation routines.

Working with array objects

- `np.ndarray`s are iterable.
- May re-use any operations that expect an iterable.
- *Could* implement element-wise math functions by computing for each element.
- But: Does not exploit `np.ndarray`'s structure hindering performance.
- Instead: Common functions form `math` module are reimplemented.
- Element-wise unary functions, e.g., `np.sin(array)`
- Element-wise binary functions, e.g., `array + array`
  - Both operands need to be of same shape.
  - When one of the operands is a scalar it is broadcasted to have the same shape as the other operand.

Note on comparison operations

- Comparison operators (`==`, `!=`, `<`, ...) are evaluated element-wise.
- Return a `np.ndarray` of booleans, i.e., a bitmask.
  - Reminder: Bitmasks may be used for array indexing.
- Python standard includes functions `all` and `any` to check if all elements or at least one element of an iterable evaluate to `True`.
- Use `np.all` and `np.any` for `np.ndarray`s.
- Use `np.array_equal` to check equality of `np.ndarray`s.

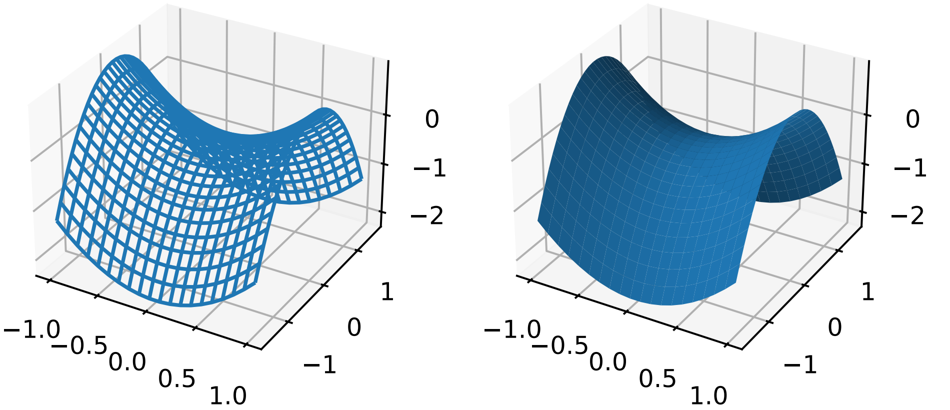
np.meshgrid

- `np.meshgrid(*xi)`
- Returns a list of coordinate matrices from coordinate vectors.
- Specifically for indices: `np.indices(dimensions)`
- Example:

```
c = ['red', 'green', 'blue']
b = [-1, +1]

cv, bv = np.meshgrid(c, b)

for comb in zip(cv.flatten(), bv.flatten()):
    print(comb)
```

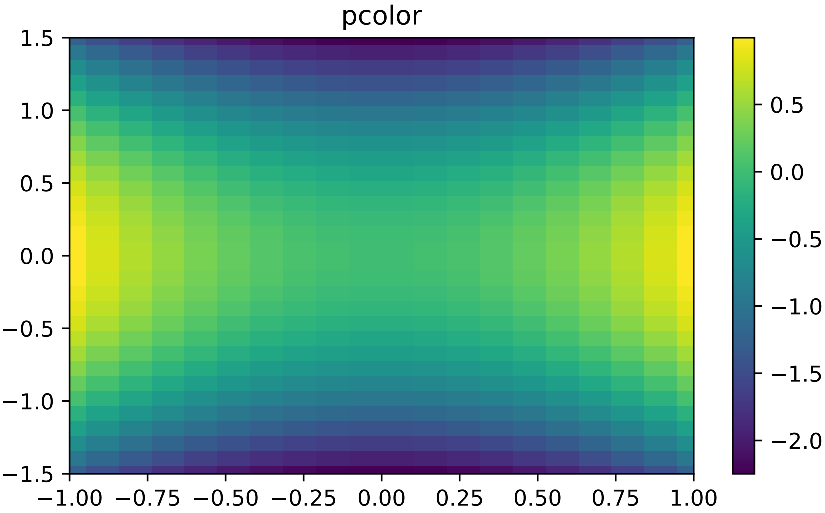
Three-dimensional surface plots

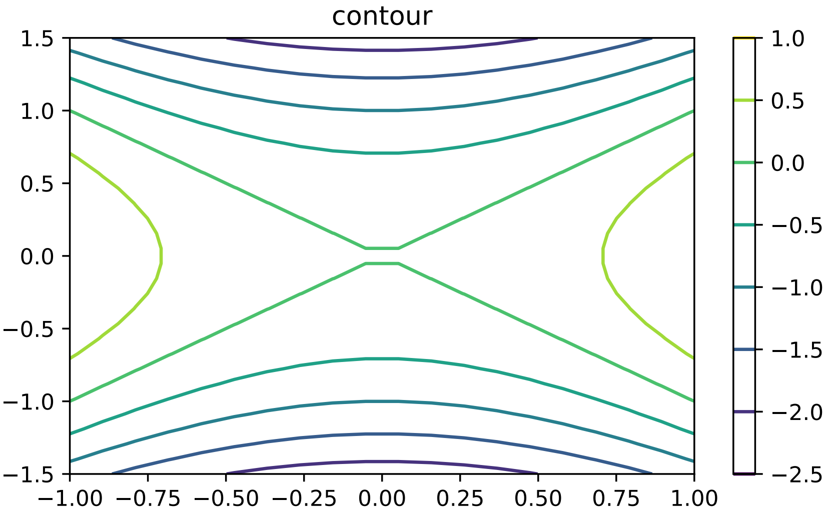
```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

X, Y = np.meshgrid(np.linspace(-1.0, 1.0, 20), \
                   np.linspace(-1.5, 1.5, 30))
Z = X**2 - Y**2

fig, (ax1, ax2) = \
    plt.subplots(1, 2, figsize = (6.4, 3.6), \
        subplot_kw={"projection": "3d"})

ax1.plot_wireframe(X, Y, Z)
ax2.plot_surface(X, Y, Z)
```


0
−1
−2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0
0
−1
−2
1
0
−1
−1.0
−0.5
0.0
0.5
1.0

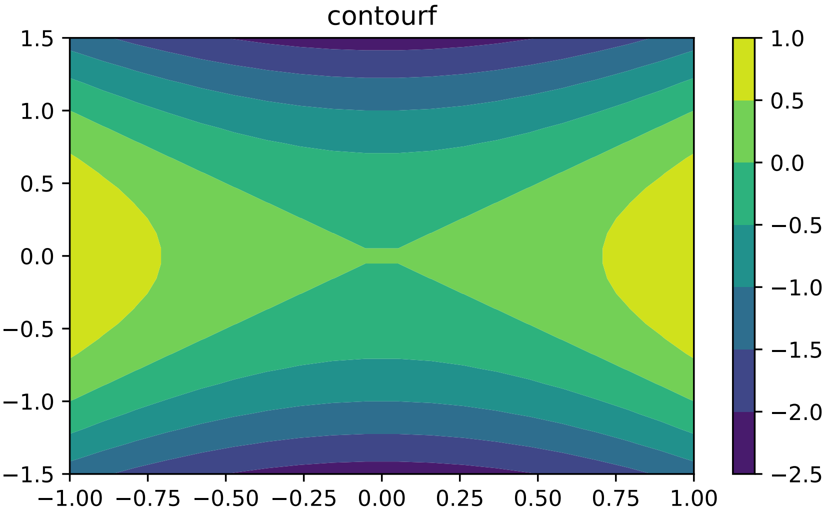
Pseudo three-dimensional plots

```python
# X, Y, Z from surface plot

fig, ax = plt.subplots(figsize = (6.4, 3.6))
ax.set_xlim(-1.0, 1.0)
ax.set_ylim(-1.5, 1.5)

ax.set_title("pcolor")
fig.colorbar(ax.pcolor(X, Y, Z, shading = 'auto'))

#ax.set_title("contour")
#fig.colorbar(ax.contour(X, Y, Z))

#ax.set_title("contourf")
#fig.colorbar(ax.contourf(X, Y, Z))
```


pcolor
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
0.5
0.0
−0.5
−1.0
−1.5
−2.0



contour
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5


contourf
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5

Modifying the shape of NumPy arrays

- `array.reshape(shape)`:
  - May not change the total number of elements.
  - Raises an error if it would change array's size.
  - `shape = -1` *flattens* the array.
- Example:

```
a = np.array([[1, 2], [3, 4], [5, 6]])
a.reshape(2, 3)     # [[1 2 3]
                    #  [4 5 6]]
```

- Side note: Internally arrays are stored linearized in contigious memory.
  - `reshape` does not change the data.
  - `reshape` only changes how the linearized index is calculated.

Modifying the shape of NumPy arrays

- `array.resize(shape)`:
  - May change the total number of elements.
  - If new size is larger fills up with zeros.
  - If new size is smaller *forgets* last elements.
- Otherwise same logical behavior as `reshape`.
- Permanence
  - `resize` changes an array permanently.
  - `reshape` does not change the array, but returns a *view* to its data.
- Copying NumPy arrays:
  - Use `copy` method of `np.ndarray`.
  - Slices return views, i.e., no copies.

Permuting dimensions of NumPy arrays

- `array.transpose(axes)`: Permutes axes.
  - `axes` is a list of integers describing the permutation.
  - The `i`-th axis of the returned array is the `axes[i]`-th of the input.
  - May return a view or an array depending on input.
  - Default value: Reverses the order.
- `array.T`: Shorthand for `array.transpose()`
  - `A.T.T == A`
- `array.swapaxes(i, j)`: Swaps axes `i` and `j` of an array.
  - May return a view or an array depending on input.

Reductions

- Operations reducing the number of elements of an array.
- Often: Computing a single value from an array, e.g., sum, average, ...
- NumPy provides optimized functions:
  - `np.sum`, `np.prod`
  - `np.mean`, `np.median`, `np.average`
  - `np.min`, `np.max`
  - `np.all`, `np.any`
- NumPy allows to specify which dimensions to reduce.
  - Optional parameter `axis`.
  - Default value: Reduce all dimensions.

Examples

```
a = np.array([[1, 2, 3],
              [4, 5, 6]])

print(np.sum(a, axis = 0))      # [5 7 9]
print(np.sum(a, axis = 1))      # [ 6 15]
print(np.sum(a))                # 21

print(a.sum(0))                 # [5 7 9]
print(a.sum(1))                 # [ 6 15]
print(a.sum())                  # 21
```

Further supported mathematical functions

- Matrix multiplication (`@`), inner and outer product, ...
- Norms of vectors, matrices and tensors.
- Determinant of a matrix.
- Computation of eigenvalues and eigenvectors.
- QR factorization and Singular Value Decomposition.
- Computation of the inverse of a matrix.
- Solving Systems of Linear Equations.
- Polynomials
- ...

Systems of Linear Equations

- Linear equations that need to be satisfied at the same time.
- Restrict ourselves to systems with as many unknowns as equations.
- Question: Which values for the unknowns solve the system of equations?
- Example:

$$\begin{aligned} x_1 + 2x_2 - 5x_3 &= 7 \\ 8x_1 - 3x_2 + 2x_3 &= 1 \\ 9x_2 - 9x_3 &= -2 \end{aligned}$$

- Form *coefficient* matrix A and vectors of constants b and of unknowns x:

$$A = \begin{pmatrix} 1 & 2 & -5 \\ 8 & -3 & 2 \\ 0 & 9 & -9 \end{pmatrix}, \quad b = \begin{pmatrix} 7 \\ 1 \\ -2 \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

- The system is then given by the matrix equation $A\,x = b$.
- Corresponding code:

```
A = np.array([[1, 2, -5],
              [8, -3, 2],
              [0, 9, -9]])

b = np.array([7, 1, -2])
```

- Use `np.linalg.solve` to solve $A\,x = b$:

```
np.linalg.solve(A, b)
```

- Solution:

$$x = \begin{pmatrix} -0.28019324 \\ -2.79710145 \\ -2.57487923 \end{pmatrix}$$

A note on comparing to zero

- Floating point values are not exact.

⇒ Never compare a computed floating-point value to another floating-point value using the equality operator `==`.

- `np.isclose(a, b)`: Element-wise equal comparison within a tolerance.

- Example: To verify solution x we might want to compare $A\,x$ to y.

```
np.all(A @ x == b)
    # Probably never true!

np.all(np.isclose(A @ x, b))
    # True withing certain tolerance.
```

Random numbers

- Submodule `np.random`
- Allows quick generation of arrays of random numbers.
- Supports different distributions:
  - `uniform`
  - `normal`
  - `laplace`
  - ...
- Functions: `np.random.<name of distribution>`
  - Different parameters depending on distribution.
  - Example: `np.random.normal(mu, sigma, shape)`

Explore the Reference

- Only a small overview could be given in this script.
- Take your time to explore the NumPy module yourself as required:
  - User Guide
  - Tutorials
  - Reference

Measuring performance

- Simplest approach: Measure time it took to run a certain task.
- Does not give an absolute value of performance, but allows for comparison.
- Computers usually have different time sources.
- Module `time` provides various time-related functions.
  - `time.time()`: Unix time, i.e., number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970.
  - `time.perf_counter()`: High-resolution with undefined reference point.
- Tasks to be measured are often very short.

  $\Rightarrow$ Measure once running multiple times and compute average.

Example

```
N = 10000; R = 2000; X = np.linspace(0, 100, N)

tic = time.perf_counter()
for run in range(R): s = sum(X)
tPython = time.perf_counter() - tic

tic = time.perf_counter()
for run in range(R): s = np.sum(X)
tNumPy = time.perf_counter() - tic

print(f"Python-Method: {tPython/R:.5e}s")
print(f"NumPy-Method:  {tNumPy/R:.5e}s")
print(f"NumPy is {tPython/tNumPy:5.1f} times faster!")
```

SymPy: Computer Algebra System

- So far: Numerical computation of mathematical expressions.
  - Floating point numbers are not exact using numerical computation.
  - Floating point arithmetics are evaluated in a certain precision.
  - ⇒ Floating point numbers are **always** approximations.
- Symbolic computation, i.e., using a Computer Algebra System:
  - Exact computation.
  - Allows expressions containing variables with no given value, i.e., symbols.

```
x = 1.0
for _ in range(20):
    x = (x - 0.9) * 10

print(x)    # ? Expected: 1.0
```

Symbolic Computations

- By convention: `import sympy as sym`
- Constants:
  - Integer: `sym.Integer(number)`
  - Rational: `sym.Rational(numerator, denominator)`
  - Float: `sym.Float(number)`
- **Note:** Arguments may be given as integer or string representing numbers.
- **Note:** Initializing a `Float` using a floating point value limits its accuracy.
- Example

```
a = sym.Integer(1)
print(a / 3)        # 1/3
print(a / 3.0)      # 0.333333333333333
```

Symbols

- `sym.symbols` function is used to create symbols.
- Name(s) of symbol(s) are given as argument(s) as
  - a comma or whitespace delimited string or
  - a sequence of strings.
- Returns an instance or a sequence of instances of `Symbols`.
- To create indexed symbols a range syntax using colon (`:`) is supported.
- Range syntax may also be used to create symbols of lexicographical range.

Examples

```
x = sym.symbols('x')
x, y = sym.symbols(['x', 'y'])

X = sym.symbols('x:3:5')          # X = (x3, x4)
X = sym.symbols('x:5')            # X = (x0, x1, x2, x3, x4)
X = sym.symbols('x2:4(3:5)')      # X = (x23, x24, x33, x34)

x, y, z = sym.symbols('x:z')

print(x / z +  y / z)               # x/z + y/z
```

Substitute symbols

- `expr.subs` method is used to substitute a symbol in an expression.
  - `expr.subs(x, y)`: Replaces `x` by `y`.
  - `expr.subs((x, y), (z, sym.pi))`: Replaces `x` by `y` an `z` by π.
  - `expr.subs({x: y, z: sym.pi})`: Same as above.
- Returns a new expression.
- Example:

```
x = sym.symbols('x')

y = (x - sym.Rational(9, 10)) * 10
for _ in range(20 - 1):
     y = y.subs(y, (y - sym.Rational(9, 10)) * 10)
print(y.subs(x, 1))    # 1
```

Mathematical functions

- SymPy supports man mathematical functions:
  - Trigonometric functions: `sym.sin(x)`, ...
  - Inverse trigonometric functions: `sym.asin(x)`, ...
  - Hyperbolic functions: `sym.sinh(x)`, ...
  - Exponential and logarithmic: `sym.exp(x)` and `sym.log(x, base)`
  - Real and Imaginary value of a complex: `sym.re(x)` and `sym.im(x)`
  - `sym.sqrt(x)`, `sym.Abs(x)`, `sym.Max(x)`, `sym.Min(x)`, ...
- Constants:
  - Imaginary unit: `sym.I`
  - Euler's number: `sym.E`
  - Pi: `sym.pi`
  - Infinity: `sym.oo`
  - Not A Number: `sym.nan`

Transformations – Simplifications

- Expanding polynomial expressions: `sym.expand(expr)`
- Factor polynomials into irreducible factors: `sym.factor(expr)`
- Collects common powers of a term in an expression: `sym.collect(expr)`
- Simplify rationals: `sym.cancel(expr)`
- Perform partial fraction decomposition: `sym.apart(expr)`
- ...
- *Simplify* an expression to its simplex form: `sym.simplify(expr)`
  - *Simplest* is not well-defined.
- Use `evalf()` method to perform numerical evaluation of an expression.

Examples

```
x = sym.symbols('x')
f = sym.cos(x) * sym.exp(x)

d = sym.diff(f, x)              # -exp(x)*sin(x) + exp(x)*cos(x)
sym.factor(d)                   # (-sin(x) + cos(x))*exp(x)

i = sym.integrate(d, x)         # exp(x)*cos(x)
i == f                          # True

sym.integrate(sym.exp(-x), (x, 0, sym.oo))     # 1

sym.limits(x / (x+1), x, sym.oo)               # 1

sym.solve(sym.sin(x), x)                       # [0, pi]
```

```
sym.exp(x).series(x, 0, 2)      # 1 + x + O(x**2)
f.series(x, 0, 2)               # 1 + x + O(x**2)

sym.exp(x).series(x, 0, 3)      # 1 + x + x**2/2 + O(x**3)
f.series(x, 0, 3)               # 1 + x + O(x**3)

y = sym.Function('y')
Y = sym.dsolve(y(x).diff(x) - y(x), y(x))
    # Eq(y(x), C1*exp(x))

Y.rhs.diff(x) == Y.rhs    # True

Y = sym.dsolve(sym.Eq(y(x).diff(x), y(x)), y(x))

y = sym.Function('y')(x)
Y = sym.dsolve(y.diff(x) - y, y)
    # Eq(y(x), C1*exp(x))
```

Refer to the official documentation

- Not all (by far) features of SymPy have been shown.
  - Custom Functions
  - Solving many different kinds of equations
  - Matrices
- See the SymPy documentation!

pandas: Data Analysis Library

- Data analysis and manipulation tool.
- Suited for different kinds of data, e.g.:
  - Tabular data with heterogeneously-typed columns, e.g., spreadsheets.
  - Ordered an unordnered time series data.
  - Arbitrary matrix data with row and column labels.
  - Statistical data sets.
- Supports loading data from different sources, e.g.:
  - Flat files, e.g., CSV, OpenDocument and Microsoft Office spreadsheets.
  - Databases and HDF5
- By convention: `import pandas as pd`
- Internally uses NumPy

Data structures

- `pd.Series`: 1-dimensional labeled homogeneously-typed array
- `pd.DataFrame`:
  - 2-dimensional labeled size-mutable tabular structure
  - homogeneously-typed rows, but potentially heterogeneously-typed columns
  - Similar to R's `data.frame`.
- Mutability of data structures:
  - Value-mutable, i.e., the contained values are mutable.
  - Not always size-mutable, e.g.,
    - Length of a Series can not be changed.
    - Additional columns can be inserted into a DataFrame.
- Data type of the elements: NumPy

Examples

```
import pandas as pd

df = pd.DataFrame({
    "First Name": ["Julian", "Monika", "Hans"],
    "Last Name": ["Huber", "Schmidt", "Meyer"],
    "Student number": [205406, 206957, 208476],
    "Grade": [3.3, 4.0, 5.0],
})

print(df)
    #   First Name Last Name  Student number  Grade
    # 0     Julian     Huber          205406    3.3
    # 1     Monika   Schmidt          206957    4.0
    # 2       Hans     Meyer          208476    5.0
```

```
grades = df["Grade"]

print(grades)
    # 0    3.3
    # 1    4.0
    # 2    5.0
    # Name: Grade, dtype: float64

g = pd.Series([2.0, 3.3, 5.0], name = "Grade")
df["Grade"] = g

print(grades.mean(), grades.std())          # 4.1 0.85
print(df["Grade"].mean(), df["Grade"].std()) # 3.4 1.50
```

- A single column of a `DataFrame` is a `Series`.
- pandas supports many functions required in statistics.

Operations on Series

- Supports element-wise operations on `Series`'.
- Syntax is the same as known from NumPy.

Adding subsets to DataFrames

- Assign a `Series` object to a column that does not exist yet.

Example

```
pl["Capital"] = pl["PricePerUnit"] * pl["Stock"]
```

Selecting subsets of DataFrames

- A single column as a `Series`: `df[column]`
- Multiple columns as a `DataFrame`: `df[[column1, column2]]`
- Filter specific rows: `df[condition]`
  - `condition` must be a `pd.Series` of booleans of length rows.
  - `pd.Series` supports method useful for filtering, e.g., `isin`.
  - `condition` may be composed of conditions using `|` (or) and `&` (and).
- Selecting specific rows and columns: `df.loc[condition, column]`

```
df[["Student number", "Grade"]]
    #    Student number  Grade
    # 0          205406    2.0
    # 1          206957    3.3
    # 2          208476    5.0

df[df["Grade"] < 5.0]
    #   First Name Last Name  Student number  Grade
    # 0     Julian     Huber          205406    2.0
    # 1     Monika   Schmidt          206957    3.3

df[df["Student number"].isin([205406, 205508])]
    #   First Name Last Name  Student number  Grade
    # 0     Julian     Huber          205406    2.0

df.loc[df["Grade"] >= 5, ["First Name", "Last Name"]]
    #   First Name Last Name
    # 2       Hans     Meyer
```

Further operations

- Sort `DataFrame` by columns using its `sortValues` method. Arguments:
  - `by`: Column or list of columns by which to sort.
  - `ascending = True`: Sort in ascending or descending order.
- Filter unset values using method `isna()`.
- Only output the first few rows using method `head()`.
- Plot your data?

  $\Rightarrow$ Use Matplotlib!
- Require mathematical function not supported by pandas directly?

  $\Rightarrow$ Use NumPy!

Reading/Writing tabular data

- Read from CSV: `pd.read_csv(filepath)`
- Write to CSV: `df.to_csv(filepath)`
- Read from Microsoft Spreadsheet: `pd.read_excel(filepath)`
  - `sheet_name`: Specify which sheet to read. Default: First
  - To read multiple sheets (as a dictionary) use `sheet_name = None`.
  - `col_index`: Specify column (by index) containing row index.
  - `header`: Specify row (by index) containing the header.
- Write to Microsoft spreadsheet:

  `df.to_excel(fpath, sheet_name = "exam", index = False)`

  - **Note:** To store multiple sheets in one spreadsheet a `pd.ExcelWriter` object must be specified instead of a file path for `fpath`.

Explore the Reference

- Only a small overview could be given in this script.
- Take your time to explore the pandas module yourself as required:
  - User Guide
  - Tutorials
  - Reference

SciPy: Scientific computing in Python

- Fundamental package used for scientific computing.
- Provides algorithms for:
  - Optimization
  - Integration and Interpolation
  - Eigenvalue problems
  - Algebraic equations
  - Differential equations
  - Statistics
- Performant!
  - Extends NumPy.
  - Wraps highly optimized implementations written in C, C++, Fortran, ...

Constants

- Direct access:
  - `scipy.pi`
  - `scipy.constants.golden` (golden ratio)
  - `scipy.constants.c`, `scipy.constants.h`, ...
  - SI units
- Access via functions:
  - `scipy.constants.value(key)`: Value of constant `key`.
  - `scipy.constants.unit(key)`: Physical unit of constant `key`.
  - `scipy.constants.precision(key)`:
    Relative precision of numerical value of constant `key`.
  - **Note:** `key` is specified as a string.
  - List of constants

Examples

```
from scipy import constants as sc

print(sc.pi)    # 3.141592653589793

print(sc.value("neutron mass"))
print(sc.precision("neutron mass"))
print(sc.unit("neutron mass"))
    # 1.67492749804e-27
    # 5.671887297281165e-10
    # kg
```

Special Functions

- Airy functions
- Elliptic functions and integral
- Bessel functions
- Struve functions
- Raw statictical functions
- Information theory functions
- Gamma and related functions
- Error function
- Fresnel integrals

- Legendre functions
- Ellipsoidal harmonics
- Orthonogal polynomials
- Hyper geometric functions
- Parabolic cylinder functions
- Mathieu and related functions
- Spheroidal wave function
- Kevin functions
- Combinatorics
- Lamber W and related functions

- ...
- List of special functions

Example

```
from scipy import special as sf

# Bessel function of 1st kind in x = 0
print(sf.jv(1, 0))    # 0.0
```

Integrals

- Numerical integration for various cases.
- Simplest case: $\displaystyle\int_a^b f(x)\,\mathrm{d}x$ with scalars a, b, x.
- `scipy.integrate.quad`:
  - Parameters: f, a, b
  - Returns a tuple of containing the integral and its error.
  - Optional parameter:
    - `args`: Further arguments fo f, e.g., `args = (y, z)` for $f = f(x, y\, z)$.
    - Accuracy
    - ...
  - **Note:** ∞ is `scipy.inf`

Example

```python
import math
import scipy
from scipy import integrate as integrate

print(integrate.quad(math.sin, 0, math.pi))
    # (2.0, 2.220446049250313e-14)

f = lambda x : math.exp(-x)

print(integrate.quad(f, 0, scipy.inf))
    # (1.0000000000000002, 5.842606742906004e-11)
```

Other Integrals

- `dblquad`: $\int_a^b \int_{g(x)}^{h(x)} f(x, y)\,\mathrm{d}x\,\mathrm{d}y$
  - See reference!

- `tblquad`: $\int_a^b \int_{g(x)}^{h(x)} \int_{q(x,y)}^{r(x,y)} f(x, y, z)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z$
  - See reference!

- `quad_vec`: $\int_a^b \vec{f}(x)\,\mathrm{d}x$
  - See reference!

Example: Double Integral

```
import numpy as np
import scipy

gaussian = lambda x, y, a : np.exp(-(x**2 + y**2) / a )
alpha = 1

print(scipy.integrate.dblquad(gaussian, \
    -scipy.inf, scipy.inf, \
    -scipy.inf, scipy.inf, \
    args=(alpha,)))

    # (3.141592653589777, 2.517308673743208e-08)
```

Optimization of Integrals

- Numerical integration: Weighted sum of discrete points
- Invocation of f for each single point
- Invocation of f can be costly.
- Alternative: *Vectorization*
  - Use NumPy functions to compute all points in a single call.
  - Pass a NumPy array of samples to the integrator.
  - Use one of many vector integrators.

Integrators with given fixed samples

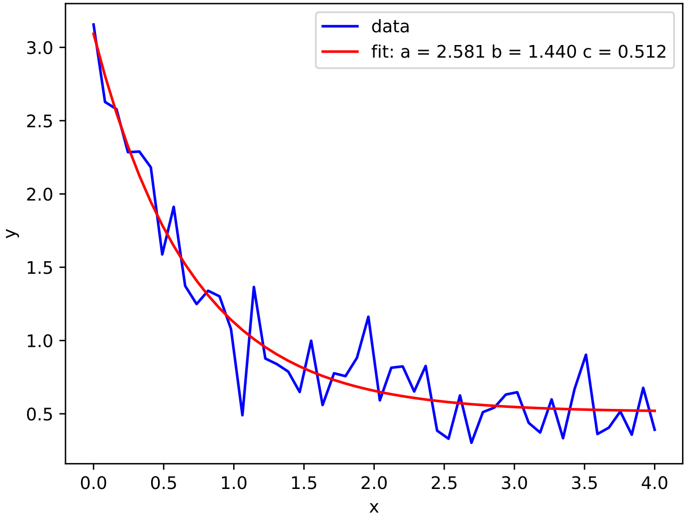
- Common Interface:
  - `Y`: Vector with pre-evaluated data points
  - `dx`: Space between two data points, i.e., the weight.
  - `axis`: Number of index along which to integrate.
- `scipy.integrate.trapz` and `scipy.integrate.simp`:
  - `X`: Vector of x-values in which `Y` was evaluated (`dx` is then ignored)
- `scipy.integrate.romb`:
  - `Y` needs to be of size $2^n + 1$ with $n = 0, 1, 2, \ldots$.
- In most cases using default `quad`rature is preferred.
- Details depend on use case.

Curve Fitting

- Input: Measured data and an assumed functional form with free parameters.
- How to determine free parameters?
- `scipy.optimize.curve_fit` with parameters:
  - Measured data as iterable.
  - Functional form as a function (or lambda) with positional parameters `x` followed by parameters to be determined.
- Returns a tuple of
  - NumPy array of determined parameters.
  - Covariance of parameters with respect to measured data.
- Reference

Example

```
from scipy.optimize import curve_fit

x = np.linspace(0, 4, 50)
f = lambda x, a, b, c: a * np.exp(-b * x) + c
y = f(x, 2.5, 1.3, 0.5)
noise = 0.2 * np.random.normal(size = x.size)
measured = y + noise

(a, b, c), pcov = curve_fit(f, x, measured)

plt.plot(x, measured, 'b-', label = 'data')

plt.plot(x, f(x, a, b, c), 'r-', \
    label = f"fit: a = {a:5.3f} b = {b:5.3f} c = {c:5.3f}")
```


data
fit: a = 2.581 b = 1.440 c = 0.512
3.0
2.5
2.0
1.5
1.0
0.5
y
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x

Common Issues

- `curve_fit` may fail or return utter nonsense.
- Often a solution: Provide an initial guess and/or limits.
- Initial guess: `p0`
  - Iterable, e.g., a tuple, of values for parameters that you *think* are reasonable close.
- Bound: `bounds`
  - Tuple with two elements: Lower and upper boundary.
  - Boundary can either be a single values (applies to all parameters) or an iterable (individual values for each parameter).
- Another approach is to limit the fit range by using only a subset of the data.

Fourier Transform

- Converts a function into a form that describes the frequencies present in the original function.
- *Fast Fourier Transform*: Optimized function that computes the Fourier transform of a signal vector.
- SciPy offers many helpers for such problems:
  - Forward and inverse transforms
  - Different norms.
  - 1-, 2-, and N-dimensional discrete transforms
  - Sine and Cosine transforms
- Reference

Notable Mentions

- Signal Processing Module
  - Convolutions and correlations
  - Tutoral
  - Reference
- Optimization Module
  - Minimization and root-finding
  - Tutoral
  - Reference

Notable Mentions

- Interpolation Module
  - Interpolation and one and multiple dimensions
  - Tutoral
  - Reference

Explore the Reference

- User Guides and Tutorials
- Reference

Computer Architectures – The Basics

- Python is designed such that neither the user nor the developer has to think about the underlying hardware.
- However, to benefit of all features of modern computer architectures, the developer requires basic knowledge of their structure.
- Hence we only focus on the basics which every programmer should know.
- Especially on how to exploit a certain feature in Python.

Processor

- A digital circuit that performs operations, i.e., *instructions*, on data.
- *Arithmetic logic unit*: Unit that performs arithmetic and bitwise operations on input operands.
- Registers to hold data or instructions.
- Other units and counters.
- A processor is working in cycles:
  - In each cycle the work *advances*.
  - Frequency: Cycles per time unit
  - ⇒ Performance is given by frequency and performed instructions per cycle.
  - Frequency and instructions per cycle can not be scaled arbitrarily.

How to increase performance?

- Increase of *bit-level parallelism*
  - Increase in size of registers.
  - Today: 64-bit registers, no trend to 128-bit architectures.
  - Nothing to do in Python.
- Increase of instruction-level parallelism
  - Parallel or simultaneous execution of a sequence of instructions.
  - Today: Superscalar architectures with more than 1 instruction per cycle.
  - Typically max 4 instructions per cycle.
  - Nothing to do in Python.

How to increase performance?

- Increase of data processed in a single instruction
  - Single instruction, multiple data (SIMD)
  - Simultaneous (parallel) computation where each unit performs the exact same instruction at any given moment on different data.
  - Today: Up to 512 bit vector registers, i.e., working on up to 8 double precision floating-point values in parallel.
  - Python: Use optimized libraries making use of SIMD, e.g., NumPy.
- Increase of thread-level parallelism.
  - Multiple threads per processor, i.e., Simultaneous multithreading (SMT).
  - Multiple processors, i.e., cores, on a single CPU.
  - Python: Multi-threading or multi-processing.

Memory Hierarchy

- Memory technologies have vastly different trade-offs between capacity, latency, bandwidth, energy and cost.

  ⇒ Hierarchy of different memory technologies

- Memory hierarchy (usually) managed by hardware and operating system.
  - All data is stored in (slow) main memory.
  - *Copies* of some data are stored in (fast) memory, i.e., caches.
  - Multiple levels of caches.
- System autonomously stores data in fast memory depending on usage pattern.
- Some systems allow programmers to influence data movement.

Memory Topology

- A cache can either be *private* to a core or *shared* by multiple cores.
- Typical modern cache topologies:
  - Core private level 1 cache
  - Core private level 2 cache or shared level 2 cache per core group.
  - Single or multiple level 3 caches shared by all cores of by group of cores.
- Typically modern main memory topologies:
  - Memory controller(s) shared by all cores.
  - Memory controller(s) per group of cores.
- All cores have access to all memory but with varying bandwidth and latency.
- Data in caches is guaranteed to be coherent (cache coherence).

Memory Access Patterns

- Implementation of caches in hardware is motivated by observation of locality:
  - Temporal locality: If a location has been accessed recently, it is likely to be accessed again, i.e., re-used, soon.
  - Spatial locality: If a location has been accessed recently, it is likely that nearby locations will be accessed soon.
- Adhere to principle of locality to achieve high performance.
- Python: Use optimized libraries, e.g., NumPy.
- Random memory access does not obey principle of locality

  $\Rightarrow$ Avoid if possible!

Cluster computing

- Multiprocessor systems
  - Computing systems with multiple connected CPUs.
  - Modern implementations (usually) rely on cache-coherent Non-uniform memory access (ccNUMA).
  - Advantage: All cores of all CPUs can access all memory.
  - Disadvantage: Accessing non-local memory suffers from higher latency.
- Distributed computing
  - A set of computers connected via high-speed network to work together.
  - Individual computers communicate and coordinate their actions by passing messages to one another.
  - Disadvantage: Message passing must be implemented by the developer.

Parallelism

- Problems can often be divided into smaller ones which can be solved in parallel.
- Use multiple parallelism levels in software to take full advantage of hardware.
- Processes vs. Threads:
  - Both are independent sequences of execution.
  - Processes run in separate memory space.
    - For low-level programming languages: MPI, ...
    - May communicate by passing messages to one another.
  - Threads run in shared memory space.
    - For low-level programming languages: OpenMP, ...

Levels of Parallelism – Recommendations

- Lowest level of parallelism: SIMD

  ⇒ The exact same operation is applied onto multiple (contiguous) data.
- Threads on same core (SMT):

  ⇒ Threads should work on same datasets.
- Threads on cores with common shared cache:

  ⇒ Threads may work on same dataset.
- Threads or processes on cores with shared memory but no common cache:

  ⇒ Processes or threads should work on distinct datasets.
- Processes on cores distributed over multiple systems:

  ⇒ Processes have to work on distinct datasets.

Parallel Programming – Reasons

- Use parallel programming to benefit from the performance improvements of modern multi-core architectures.
- I/O bound problems may benefit from using parallel programming even on single-core systems.
  - I/O typically is a blocking routine, i.e., execution of other tasks is blocked while waiting for data.
  - Think about slow devices, like classical HDDs, network services, ...
  - However, the load on processing units is low.
  - May use these resources for other tasks while waiting for data.

Parallel Programming – Python

- Python provides various ways to run multiple threads or processes.
- Easiest way: Use optimized libraries, e.g., NumPy.
  - Might require changing NumPy backend.
  - `np.show_config()`
- For using multiple processes: `multiprocessing`
- For using multiple *threads*: `multithreading`.
  - A Python *thread* does not confirm to the common definition.
  - All Python *threads* run on the same hardware thread, i.e., may not benefit from multiple cores and simultaneous multithreading.
  - Python interpreter switches between Python *threads*.
  - Background: A limitation of the interpreter.

Parallel Programming – Pitfalls

- Assume two or more routines are running in parallel.
- All routines share a common resource, e.g., a variable `x`.
- Now one routine wants to read `x`.
- At the same time another routine wants to write `x`.
- Which routine accesses `x` first?
- This is referred to as a *race condition*.
- Even more complicated when `x` is a composite structure, e.g., a class object.

  ⇒ Need to implement looks, know as *mutexes*, i.e., mutual exclusion.
- **Note:** Python `multiprocessing` and other modules may save you from some race conditions, but not all of it.

multiprocessing

- `multiprocessing` excepts tasks of processes to be defined as functions:
  - E.g., `def subroutine(*args, **kwargs)`.
  - Usually these functions shall have no return value.
  - May have return values, but we do not have time to explain how it works.
- Each process itself is an object of type `multiprocessing.Process`:

```
proc = multiprocessing.Process(
        target = subroutine,
        args = tupleOfPositionalArguments,
        kwargs = dictOfKeywoedArguments)
```

- Call the method `start` of a `Process` to make the process run *asynchronously*.
- Call the method `join` of a `Process` to wait unitl the process has finished.

```python
def worker(name):
    print("Started to work:", name)
    time.sleep(1)
    print("Finished work:", name)

processes = []
for name in ["Elton", "Olaf", "Friedrich"]:
    processes.append(multiprocessing.Process( \
        target = worker, args = (name,)))

tic = time.time()
for proc in processes: proc.start()
for proc in processes: proc.join()
print("Time elapsed:", time.time() - tic)
```

⇒ A lot of time saved, but results can and will interfere with one another!

Parallel Programming – Locking Resources

- To avoid interference at critical points let one process *own* a critical resource.
- Implemented as a class: `multiprocessing.Lock`
  - Objects of type `Lock` may only be *owned* by one process at a time.
- Use method `acquire` to obtain ownership
  - Tries to get ownership of a lock.
  - If successful: Continues execution
  - Otherwise: Waits until the current owner gives up the ownership of a lock, i.e., *releases* the lock.
- Use method `release` to give up the ownership.
  - Allows others to acquire the lock.

```python
def worker(name, lock):
    lock.acquire()
    print("Started to work:", name)
    lock.release()

    time.sleep(1)

    lock.acquire()
    print("Finished work:", name)
    lock.release()

lock = Lock()
processes = []
for name in ["Elton", "Olaf", "Friedrich"]:
    processes.append(multiprocessing.Process( \
        target = worker, args = (name, lock)))
```

$\Rightarrow$ Non-critical parts run in parallel, problematic parts forced to run sequentially.

Parallel Programming – Pitfalls

- Wait for all processes to be finished before your main process terminates.
  - Otherwise remaining process may become *zombies*.
- Deadlocks:
  - It is possible to write code where processes wait for each other with none of them progressing. This is called a *deadlock*.
  - Most often this happens when multiple locks, or similar mechanisms, are in play at the same time.

  ⇒ As far as possible, use only one lock mechanism at a time!
  - If your run into a deadlock: Terminate by pressing `Ctrl+C` in your terminal or kill the process using other means, e.g., a task manger.

Communication between processes – Part 1

- `multiprocessing.Value`:
  - A single variable to be shared between processes.
  - Build-in lock.
  - Fixed data type, e.g., `num = Value('i'), 42)` is a shareable `int`.
  - Access value of the variable using attribute `value`, e.g., `num.value`.
- `multiprocessing.Array`:
  - A shareable C-style, i.e., NumPy like, array.
  - Example: `arr = Array('i', range(5))`
  - Access values like in a list, e.g., `arr[0]`

Communication between processes – Part 2

- `multiprocessing.Queue`:
  - A collection of objects maintained in a sequence.
  - Can be modified by:
    - Addition of objects at one end.
    - Removal of objects from the other end.

    ⇒ A first-in-first-out (FIFO) data structure.
  - Use method `put` to add an object to a queue.
  - Use method `get` to obtain an object of a queue.
    - Waits if queue is empty, i.e., might cause a deadlock.
    - Possible to set a timeout value.
  - May be used to distribute work over processes, i.e., a work queue.

```python
def producer(ID, queue, stop):
    while not stop.value:
        queue.put(f"Stuff from producer #{ID}")
        time.sleep(.1)

def consumer(ID, queue, stop):
    while not stop.value:
        if not queue.empty():
            goods = queue.get()
            ...
        time.sleep(.1)

queue = multiprocessing.Queue()
stop = multiprocessing.Value('b', False)
```

```python
processes = []
for i in range(5):
    processes.append(multiprocessing.Process( \
        target = producer, args = (i, queue, stop)))

for i in range(10):
     processes.append(multiprocessing.Process( \
         target = consumer, args = (i, queue, stop)))

for p in processes: p.start()
time.sleep(10)
stop.value = True
for p in processes: p.join()

print("Unconsumed goods:", queue.qsize())
```

Communication between processes – Part 3

- `multiprocessing.Pipe`
  - Sets up a duplex, i.e., two-way, communication between two processes.
  - Returns a tuple of two `multiprocessing.Connection`s representing the two ends of the pipe.
  - Use connection's `send` and `recv` methods to *transfer* data.
    - `recv` is a blocking routine, i.e., it waits for data.
  - **Note:** Data in a pipe may become corrupted if two processes (or threads) try to read from of write to the **same** end of the pipe at the same time.

```
def england(conn):
    conn.send([42, None, 'hello'])
    conn.close()

def france(conn):
    print(conn.recv())
    conn.close()

folkstone, coquelles = multiprocessing.Pipe()
p = multiprocessing.Process( \
    target = england, args = (folkstone,))
q = multiprocessing.Process( \
    target = france, args = (coquelles,))

p.start()
q.start()
p.join()
q.join()
```

Launching parallel tasks – A higher level approach

- The `concurrent.futures` module provides a high-level interface for asynchronously executing callables.
- The asynchronous execution can be done using *threads* or processes.
- Same interface, but different executors:
  - *Threads*: `ThreadPoolExecutor`
  - Processes: `ProcessPoolExecutor`.
- Reference

Example

```python
import concurrent.futures
import math

def is_prime(n):
    if n < 2: return False
    if n == 2: return True
    if n % 2 == 0: return False

    for i in range(3, int(math.floor(math.sqrt(n))) + 1, 2):
        if n % i == 0:
            return False
    return True
```

```
numbers = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

with concurrent.futures.ProcessPoolExecutor() as executor:
    for number, prime in zip(numbers, \
            executor.map(is_prime, numbers)):
        print(f"{number} is prime: {prime}")
```

Concurrency

- So far we used multiprocessing to spread tasks over available compute resources, i.e., processing units.
- This is beneficial for compute-bound tasks, i.e., tasks for which runtime is dominated by computational operations.
- We also mentioned threading as a means to speed up I/O bound tasks.
- This can be done using `multithreading` or `ThreadPoolExectutor` to *concurrently* run tasks, i.e., multiple tasks running in an overlapping manner.
- This is referred to *concurrency*.
- **Note:** Parallelism is a form of concurrency.

asyncio – Asynchronous I/O

- Introduced in Python 3.4, i.e., still relatively new.
- Allows to run *coroutines*, i.e., asynchronous routines, *asynchronously*.
- Comes with two new keywords: `async` and `await`
- Used for I/O-bound and network tasks.
- Used as a foundation for asynchronous computing.
- **Note:** `asyncio` uses *cooperative multitasking*:
  - Coroutines can be scheduled concurrently, but they are not inherently concurrent, i.e., only one coroutine is running at a time.
  - Coroutines are able to *pause* while waiting for progress and let other coroutines run in the meantime.
  - *Asynchronous code gives the look and feel of concurrency.*

Coroutines

- A function that can suspend its execution before reaching return.
- Specialized version of a generator function.
- Syntax: `async def function(...):`
- It can indirectly pass control to another coroutine for some time.
- Syntax: `await coroutine(*args)`
- `await` passes function control back to the event loop.
- Use `asyncio.run` function to run the top-level entry point coroutine.
  - Starts an event loop which is monitoring coroutines, taking feedback which coroutines are idling and scheduling coroutines.

Example

```python
import time
import asyncio

async def work():
    print("Started to work")
    await asyncio.sleep(1)
    print("Finished work")

async def main():
    await asyncio.gather(work(), work(), work())

tic = time.time()
asyncio.run(main())
print("Time elapsed:", time.time() - tic)
```

Example – Local file I/O

```python
import asyncio
import aiofiles

async def main():
    async with aiofiles.open(filename) as f:
        async for line in f:
            print(line)

asyncio.run(main())
```

- `async for` does not cause the iteration to run concurrently.

  ⇒ Iterations are especially not run in parallel.

- `async for` and `async with` provide the functionality of their synchronous counterparts with the ability to give up control to the event loop.

Third-party packages based on Asynchronous I/O

- HTTPX: HTTP client
- AIOHTTP: HTTP client and server
- aiofiles: File I/O
- websockets: WebSocket client and server
- AsyncSSH: SSHv2 client and server
- aioserial: Serial Port Extension
- asyncio subprocesses: Create and manage subprocesses

User Interfaces

- So far communication between users and our programs happen in a terminal.
- While this may be fine for developers, the general user cannot be expected to even know what a terminal is.
- How to handle these users?

⇒ Graphical User Interface (GUI)

- In Python different solutions to provide GUIs exist.
- In this course we only present tkinter.

tkinter

- Abbreviation for *Tk interface*.
- tkinter is the standard Python interface to the Tcl/Tk GUI toolkit.
- Basic idea is to prepare the GUI before showing a window.
- Once shown execute predefined functions and methods on user interaction.

⇒ Everything needs to be in functions.

- Main window is represented by class `tkinter.Tk`.
- Method `mainloop` of `tkinter.Tk`:
  - Draws the main window on screen.
  - Start an infinity loop waiting for user interaction.

Example

```
from tkinter import *

root = Tk()
root.title("Hello World")
root.geometry("600x400+500+200")

root.mainloop()
```

- Method `title` of `tkinter.Tk` sets application title.
- Geometry may be changed using method `geometry`:
  - First two numbers set size, second two number set position.
- Widgets may be manipulated in many ways.

Widgets

- May different widget exist to show text, buttons, picture, ...
- A short list of important widgets:
  - `Label`
  - `Button`
  - `Entry`
- Use a widgets `pack` method to add it to a window.
- Use classes from `tkinter.ttk` to get themed widgets.
- May replace the basic ones by themed ones using imports:

```
from tkinter import *
from tkinter.ttk import *
```

Example

```
from tkinter import *
import tkinter.ttk as ttk

root = Tk()
root.title("Widgets example")

label = ttk.Label(root, text = "This is some text")
label.pack()

button = ttk.Button(root, text = "Exit", \
    command = lambda: root.destroy())
button.pack()

root.mainloop()
```

Organize widgets – Layout

- Using `pack` widgets are appear in order as coded.
- Sometimes we need to be able to specify a particular order.
- This may be achieved using a grid to organize widgets within a window.
- Using a grid, it is not allowed to use `pack`.
- Set position by specifying row and column.
- Size may be set using row- and column-span.

```python
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()

labels = [ttk.Label(root, text = f"Text {i}").grid( \
    row = i, column = 0) for i in range(7)]

var = [tk.StringVar() for i in range(7)]
entries = [ttk.Entry(root, textvariable = var[i]).grid( \
    column = 1, row = i) for i in range(7)]

button = ttk.Button(root, text = "Exit", \
    command = lambda: root.destroy())
button.grid(row = 7, columnspan = 2)

root.mainloop()
user_input = [item.get() for item in var]
```

Basic Idea

- Use widgets, e.g. `Entry`, to allow user input.
- Specify functions as parameter `command` to buttons to process user input.

Useful functions

- `tkinter.simpledialog.askstring(title, prompt, **kw)`
- ```
  tkinter.messagebox.showinfo( \
          title = None, message = None, **options)
  ```
- `tkinter.scrolledtext.ScrolledText(master = None, **kw)`
  ```

Source Code Management

- So far: Only small programs which can easily be managed.
- Reality:
  - Projects consist of multiple source code files.
  - Multiple developers work on a project in parallel.
  - New developers come in, others leave.
  - May need to maintain different versions of a project in parallel.
- Naive approach:
  - Encode version information in filename.
  - Exchange new version with collaborators, e.g., via e-mail.
  - What happens when developers update the same file at the same time?

Version Control Systems

- Naive approach not a solution for any serious software project.
- Use a version control system to help ease the process.
- General idea:
  - A version control system is tracking changes of all files.
  - Allows us to go back in time to any previous state.
  - Full history is exchanged with collaborators.
- Two approaches:
  - Centralized: Clients push changes to one central version of the codebase.
  - Distributed: The complete codebase, including its full history, is mirrored on every developer's computer.

Git

- A distributed version control system.
- Open-source software (GPL-2.0)
- Basis for services like GitHub and GitLab.
- Was created for the development of the Linux kernel.
  - Most kernel developers are used to work within a terminal.
  - $\Rightarrow$ Git is also designed to be used from a terminal.
  - Programs allowing git to be used via GUI exist.
- Designed for collaboratively developing source code.
- May be used for collaboratively or version control of any files.
  - $\Rightarrow$ Works best with files which are plain text files.

Git Commands – Part 1

- Before doing anything let git know who you are:
  - `git config --global user.name <your name>`
  - `git config --global user.email <your email>`
- Initialize an empty local repository:
  - In current working directory: `git init`
  - In specified directory: `git init <directory>`
- Get a local copy of an existing repository:
  - `git clone <link to repository>`
  - Creates a new directory with project name in current directory.
  - Remote repository is (by default) referred to as `origin`.

Git Commands – Part 2

- Get status information of *current* repository:
  - `git status`
  - *Current* repository is determined by current working directory.
- Adding changes to the *staging* area:
  - Add all changes of a file or directory: `git add <path>`
  - Interactively ask which changes of a file to add: `git add --patch <path>`
- Commit staged changes to your repository:
  - `git commit`
  - Opens a text editor asking for a commit message.
- Show history: `git log`

Git Commits – Guidelines

- Commit messages consist of subject (first line) and a body.
- Very similar to an e-mail (on purpose).
- Write them like an e-mail to all developers explaining the changes.
- Commit messages are the documentation **why** you made the changes.
- Invest time into good commit messages!
- What are good commits?
  - Include only small changes.
    $\Rightarrow$ Larger changesets are split into many small commits.
  - Are conceptually separated.
  - Shall only include working code.
  - Shall only include source files.

Git Commands – Part 3

- Show changes to your tracked files: `git diff`
  - By default: Differences between working directory and staging area.
  - `git diff --staged`: Changes added to staging area.
- Undo local changes:
  - Undo local changes: `git restore <path>`
  - Remove changes from staging area: `git restore --staged <path>`
- Show changes of a specific commit:
  - `git show <commit>`
  - Commits are identified by an unique hash.
  - Get hash by, e.g., using `git log`.

Basic workflow

Work on your source code

$\Downarrow$

`git diff` / `git status`

$\Downarrow$

Stage your changes: `git add`

$\Downarrow$

Commit changes: `git commit`

$\Downarrow$

Repeat

Branches

- Branches store different versions of a project.
- Allows parallel development:
  - Implement new features
  - Fix bugs
  - Try out something new
- Internally handled by pointers to commits.
- A default branch is created on initialization (`main` or `master`).
- Development is usually done on other, feature, branches.

Git Commands – Part 4

- Create a new branch: `git branch <name>`
- List local branches: `git branch`
- List remote branches: `git branch -r`
- List local and remote branches: `git branch -a`
- Delete a branch: `git branch -d <name>`
  - Deletes a branch only if changes are included in another branch.
  - Use `git branch -d -f <name>` to delete anyway.
- Switch to an existing branch: `git checkout <name>`
  - Only works on *clean working tree*, i.e., no staged but uncommitted changes.
- Shortcut to create and checkout new branch: `git checkout -b <name>`

Version management

- Remember: Internally branches are just commits.

  ⇒ One may also check out a certain commit to go back in history.

- The currently checked out branch/commit is referred to as `HEAD`.
- Branches may be *merged* to apply changes from one branch to another.
  - `git merge <branch>`
  - Changes the checked out branch.
  - Usually retains the full history of changes.
  - Smart automatic three-way merges.

    ⇒ Works very well with text files, only limited with binary files.

- Branches may be *rebased* to apply changes later applied to the original branch:
  `git rebase <branch>`

Interacting with remotes

- Local branches may diverge from remote branches.
- Pull changes from remote: `git pull <remote> [<branch>]`
  - Applies changes to checked out branch.
  - May optionally specify strategy how to apply changes.
  - Download changes without applying: `git fetch origin`
- Push changes to remote: `git push <remote> [<branch>]`
  - Pull remote changes before you try to push your local changes!

Other git commands

- There are many more git commands available.
- Get help about git itself:
  - `git help`
  - `git --help`
  - `man git`
- Get help about a certain git command:
  - `git help <command>`
  - `git <command> --help`
  - `man git-<command>`

Continuous Integration and Continuous Delivery

- Automation added on top of git, e.g., by GitHub or GitLab.
- Continuous Integration:
  - Run defined tests after every source code change to detect issues.
- Continuous Delivery:
  - Automatically deploy software after continuous integration passed.

Git isn't just for software development!

Project