15-110 Principles of Computing – F19

Lecture 18: Functions 2, Modules

Teacher:
Gianni A. Di Caro

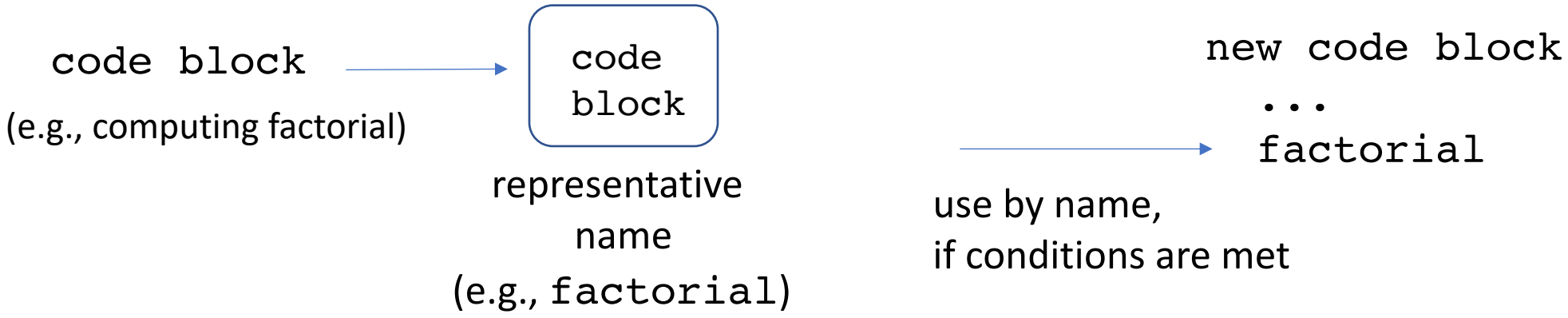
Functions: organizing the code, putting aside functionalities

- Functions are a fundamental way to *organize* the code into **procedural elements** that can be *reused*
- Functions provide **structure and organization,** that facilitate:

Fundamental ingredients in the design of computational solutions

Abstraction

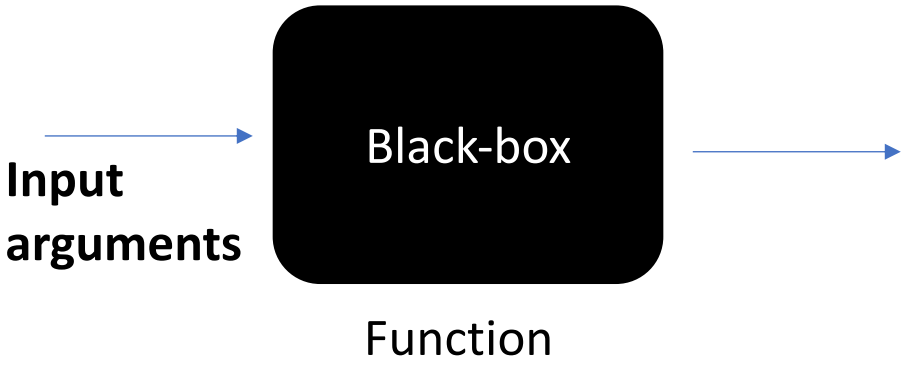
- **Abstraction** in computing aims to **hide details that are not necessary in a given context**, preserving only the information that is relevant in the context:
  - Operationally, abstraction is the process that:
    1. allows to take/write a *piece of code* (including anything relevant)
    2. *name it*
    3. use it as it were a *black-box* as long as the *conditions* for using the code are met



Abstraction types include **functions**, **classes** (abstract data types), …

Abstraction using functions

- When we define a function we are performing an **abstraction**: we take a piece of code, including objects and expressions, we name it, and in principle we can we use it, without caring about *how* the outputs are precisely obtained in the function body



Outputs

- returned values
- modifications that propagates outside the function (changes to input parameters that are mutable data types)
- side effects on the program environment

Abstraction using functions

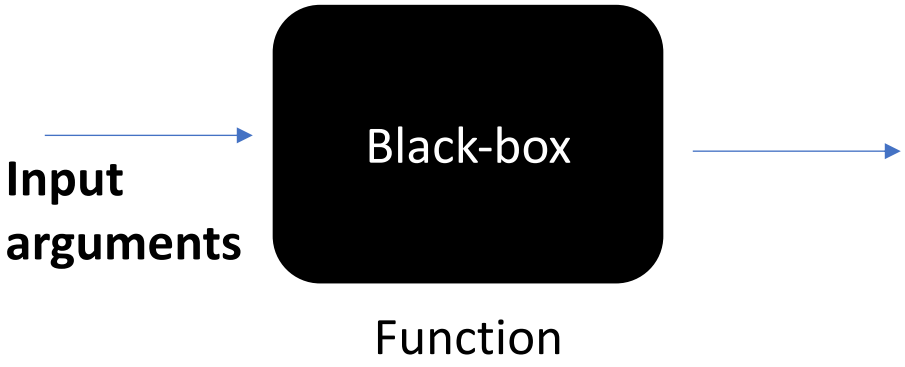
```
def make_product(values):
    prod = 1
    for v in values:
        prod *= v
    return prod
```

```
my_data = [2,4,5]
p = make_product(my_data)
s = sum(my_data)
print('Product:', p, 'Sum:', s)
```

```
def sort_and_get_median(l):
    l.sort()
    return l[len(l)//2]
```

```
my_data = [2, 4, 5, -1, 3]
median = sort_and_get_median(my_data)
min_val = my_data[0]
max_val = my_data[len(my_data)-1]
print('Min:', min_val, 'Max:', max_val,
'Median:', median)
```

Abstraction using functions



Outputs

- returned values
- modifications that propagates outside the function
- side effects on the program environment

- The only information relevant to use a function :
  - **input parameters**: types and admitted values
  - **returned objects**: types, range of values, adopted conventions
  - **description of what the function does**, including possible side effects

The precise details about *how* processing is performed are hidden by the abstraction

Example of two functions doing the same task but in different way

Find the (one) value of x that satisfies the equation $f(x) = 0$, for $x \in [a, b]$

```
def find_root(f, a, b, N):
    if f(a)*f(b) >= 0:
        print("Secant method fails.")
        return None
    a_n = a
    b_n = b
    for n in range(1,N+1):
        m_n = a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))
        f_m_n = f(m_n)
        if f(a_n)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
        elif f(b_n)*f_m_n < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == 0:
            print("Found exact solution.")
            return m_n
        else:
            print("Secant method fails.")
            return None
    root = a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))
    return root
```

```
def find_root(f, a, b, N):
    if f(a)*f(b) >= 0:
       print("Bisection method fails.")
       return None
    a_n = a
    b_n = b
    for n in range(1,N+1):
        m_n = (a_n + b_n)/2
        f_m_n = f(m_n)
        if f(a_n)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
        elif f(b_n)*f_m_n < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == 0:
            print("Found exact solution.")
            return m_n
        else:
            print("Bisection method fails.")
            return None
    root = (a_n + b_n)/2
    return root
```

Houston we have a problem!

- The (basic) abstraction through a function code only provides (exposes) in an *explicit way*:
  - **input parameters**: names
  - **returned objects**: names / expressions
  - **description of what the function does:** name of the function

```
def quadratic_roots(xx, x, c):
    x1 = -x / (2 * xx)
    x2 = math.sqrt(x**2 - (4 * xx * c)) / (2 * xx)
    return (x1 + x2), (x1 - x2)
```

$$ax^2 + bx + c = 0$$

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- I want to **use the function** (possibly developed by somebody else)
- I want to be a **client** for the numerical *services* offered by the function

- What is the precise meaning of those input parameters?
- Can I set them to any numeric value?
- What about the outputs?
- Will I always get an admissible numeric result?

Specifications

Parameters and names of the abstraction aren't enough!

We also need to **describe the abstraction**, the function and its elements in our case, in order to properly use it and reuse it!

- **Specification:** defines a *contract* between the provider/implementer of an abstraction (a function) and those who will be using the abstraction (the function), the users
- **User** ↔ **Client** for the services provided by the abstraction (the function)

Specifications: example

Assumptions: what the client has to do to use the function

Guarantees: what the provider guarantees about outputs and effects of the function

Numeric examples: show function usage and provide basic test cases

```
def find_root(f, a, b, N):
    '''Approximate solution of f(x)=0 on interval [a,b] by the bisection method.

    Parameters
    ----------
    f : function
        The function for which we are trying to approximate a solution f(x)=0.
    a,b : numbers
        The interval in which to search for a solution. The function returns
        None if f(a)*f(b) >= 0 since a solution is not guaranteed.
    N : (positive) integer
        The number of iterations to implement.

    Returns
    -------
    x_N : number
        The midpoint of the Nth interval computed by the bisection method. The
        initial interval [a_0,b_0] is given by [a,b]. If f(m_n) == 0 for some
        midpoint m_n = (a_n + b_n)/2, then the function returns this solution.
        If all signs of values f(a_n), f(b_n) and f(m_n) are the same at any
        iteration, the bisection method fails and return None.

    Examples
    --------
    >>> f = lambda x: x**2 - x - 1
    >>> bisection(f,1,2,25)
    1.618033990263939
    >>> f = lambda x: (2*x - 1)*(x - 3)
    >>> bisection(f,0,1,10)
    0.5
    '''
```

... program code continue in the next slide

Specifications: example

```
if f(a)*f(b) >= 0:
        print("Secant method fails.")
        return None
    a_n = a
    b_n = b
    for n in range(1,N+1):
        m_n = a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))
        f_m_n = f(m_n)
        if f(a_n)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
        elif f(b_n)*f_m_n < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == 0:
            print("Found exact solution.")
            return m_n
        else:
            print("Secant method fails.")
            return None
    root = a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))
    return root
```

Specifications: for every product we trade or share

SHARP

Full-Auto Electric Washing Machine

Model: ESS119 / ESS159

INSTRUCTION MANUAL

- Applying the Air Bubble Washing System.
- Applying One-Touch Operation System.
- Realizing calm washing through the innovational low-noise design.
- Before operating this washing machine, please read this instruction completely. Thank you.

Air Bubble Washing machine System

- Your washing machine is equipped with the bubble generator that generates a great deal of air bubble.
- The air bubbles knock the dirt out of the clothes during wash and they push up the heavy dirt to remove it during rinse.
- Because the double washing method using both air bubble & water current is applied to this washing machine, wash efficiency is excellent in comparison with the original washing machine system.

CONTENTS Page

A NOTE TO YOU

Thank you for buying a SHARP appliance.
SHARP washing machine are easy to use, save time, and help you manage your home better.
This manual contains valuable information about how to operate and maintain your washing machine properly and safely.
Please read it carefully.

Juiced Bikes / 1085 Bay Blvd, Suite B / Chula Vista, CA 91911
Tel: 1-888-303-8889 / www.JuicedBikes.com / mail@juicedbikes.com

OceanCurrent

Component	Specifications
Frame	6061 Aluminum heat treated / integrated & removable down-tube battery
Frame Size	Standard Frame / Step Thru Frame
Colors	Black (Flat) / Red / Greenery / Sea Foam / Army Green
Performance	24 mph (38 km/h) / Torque-based pedal assist / Throttle 20 mph 32 km/h
Battery / Standard	48V x 8.8 Ah LG 18650 / 52 Cells x 2,200 mAh / 3C
Battery / Extended Range	48V x 10.4 Ah Samsung 18650 / 52 Cells x 2,600 mAh **(Option)**
Battery / Super Extended Range	48V x 17.4 Ah Panasonic 18650 / 78 Cells x 2,900 mAh / 3C **(Option)**
Battery / Super Extended Range	48V x 21.0 Ah Panasonic 18650 / 78 Cells x 3,500 mAh / 3C **(Option)**
Motor	Bafang (8Fun) 500W / Geared hub motor / Shimano cassette
Torque Sensor	Precision Torque Sensor + HD cadence sensor up to 104 poles per revolution
Controller	6 Transistor / Sine wave / Customized tuning algorithm
Throttle	Thumb throttle included / Right or Left side mountable / 20mph - 8A limited
Display	King Meter T320 / Customized / 5 Levels assist: ECO, 1, 2, 3 and SPORT (24 mph) / LCD **(Option)**
Charger	Modary 2A / UL Certified / On-bike and off bike charging / 3.5 hour recharge time
Wire Harness	1-to-4 Quick Connect / Stainless Steel lock-ring / Water proof
Brakes	Shimano BR-M375 Mechanical Disc Brakes
Wheels and Tires	Kenda Small Block 8 / 26" / 36H
Transmission	Shimano Acera 8 speed 11-32T / Front chain ring: Pro Wheel 48T / 170mm / Chain: KMC
Fork	Aluminum front fork / Disc Brake Compatible
Spokes	Stainless / Front: 13G / Rear: 12G
Handle Bar and Stem	Cruiser wide / Quill Stem
Seat Post and Seat	Promax 27.2 mm / Quick release / Velo comfort wide
Mud Guards & Rear Rack	Compatible with industry standards
Pedals	Welgo platform pedals
Warranty	1 Year coverage on electrical components 2 Year coverage on mechanical Lifetime coverage on frame
Packing Dimensions	54" x 34" x 12" - with front wheel removed / Bike + Box: 56 lbs (25.4 kg) R2017-04-17

Glen-Gery

PRODUCT PROFILE

Revised 7/2015

Glen-Gery Extruded Brick

General

Glen-Gery manufactures many sizes of extruded bricks in a multitude of shades and textures to accommodate the visual requirements of most projects. The more popular extruded bricks have a nominal four inch bed depth. These extruded units are often referred to as cored, stiff mud, or wirecut bricks. To differentiate between wirecut bricks and wirecut finishes, Glen-Gery refers to the wirecut finish as a velour texture.

Unit Specifications

Glen-Gery extruded bricks are typically manufactured to conform to the requirements of American Society for Testing and Materials (ASTM) Standard Specification C 216, Grade SW, Type FBS and all grades of ASTM C 62. In some instances brick are manufactured to conform to ASTM C652 which includes increased core volume. These products also conform to the requirements of ASTM C 216, Grade MW. Certain products meet the requirements of ASTM C 216, Type FBX, ASTM C 902, ASTM C 652, or ASTM C 32. Inquiries should be made for specific applications or conformance to standards other than ASTM C 216 or C 62. When specifying this product, the specifications should cite:

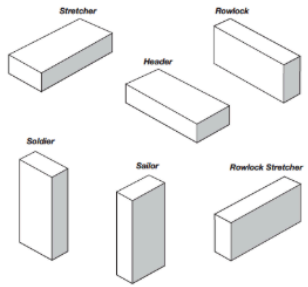
1) The product name and state "as manufactured by Glen-Gery Corporation."

2) Conformance to the requirements of the appropriate standard, typically, ASTM C 216 or C652).

3) The actual unit dimensions listed as thickness x height x length.

Example: Glenrose Battlefield as manufactured by Glen-Gery Corporation to conform to the requirements of ASTM C 216, Grade SW, Type FBS. The units shall have dimensions of 3-5/8" X 2-1/4" X 7-5/8".

Brick Positions in a Wall



1

The specification contract

- **Specification:** defines a *contract* between the provider/implementer of an abstraction (e.g., function) and those who will be using the abstraction (e.g., the function), the users / clients

- The specification contract has *two mandatory parts*:

  - ✓ **Assumptions:** describe the conditions that must be met by the users of the function
    - What are the admissible types and values / ranges of the input parameters
    - What other functions must have been executed first
    - What is supposed to be already in place before calling the function
    - …

  - ✓ **Guarantees:** describe the conditions that must be met by the function (given that the invoking conditions have been satisfied), what the providers ensures about the function
    - What are returning types and/or range of values
    - What are the side effects (e.g., on the input parameters) caused by the function
    - Under which conditions the function fails or may fail
    - …

The moral is …

Functions (methods, classes) need to be documented,
accompanied by a specification for the client

Specifications fully support abstraction and facilitate / make it possible code reusability, maintenance, and sharing

docstrings for providing specifications

- **docstring:** *documentation strings* (or docstrings) provide a structured way for providing **specifications** within functions, modules, classes, and methods

- A docstring is defined by including a **string constant** as the first statement in the object's definition

```
def add(x, y):
    "Sum two objects x and y and returns the sum."
    return x + y
```

```
def add(x, y):
    'Sum two objects and returns the sum.'
    return x + y
```

```
def add(x, y):
    '''Sum two objects and returns the sum.'''
    return x + y
```

''' triple (single) quote delimiters are the first choice for docstrings, since they allow to write **multiline descriptions**

```
def add(x, y):
    '''Sum two objects and returns the sum based on their data types.
       If the + operator cannot be used for the two objects, an error is thrown.'''
    return x + y
```

docstrings for providing specifications: `help(f)` function

- `help(f)` function: returns the docstring information about the function

```
def add(x, y):
    '''Sum two objects and returns the sum based on their data types.
       If the + operator cannot be used for the two objects, an error is thrown.'''
    return x + y
```

```
help(add)
Help on function add in module __main__:

add(x, y)
    Sum two objects and returns the sum based on their data types.
    If the + operator cannot be used for the two objects, an error is thrown.
```

docstrings for providing specifications: `help(f)` function

- Only the string constant following the definition is returned by `help()`!

```
def add2(x, y):
    '''
    Sum two numeric data types.
    Input: x and y must be numeric data types and/or boolean. Type can be mixed.
    Output: The sum of x and y is returned as an object whose data type is
            defined by the data type of the two inputs based on python's rules.
            If x and y aren't both numeric or boolean, None is  returned.
    '''
    if (type(x) != int and type(x) != float and type(x) != bool) or \
        (type(y) != int and type(y) != float and type(y) != bool):
        return None
    '''The provided inputs are fine, the + operator can be applied'''
    return x + y
```

docstrings for providing specifications: `help(f)` function

```
help(add2)
Help on function add2 in module __main__:

add2(x, y)
    Sum two numeric data types.
    Input: x and y must be numeric data types and/or boolean. Type can be mixed.
    Output: The sum of x and y is returned as an object whose data type is
           defined by the data type of the two inputs based on python's rules.
           If x and y aren't both numeric or boolean, None is  returned.
```

- The comment strings other than the first one are not displayed, being not part of the specification

Of course comments to well document the code inline are most welcome / necessary!

Inline comments: explaining the *how* part of the code

- Inline comments describe ***how*** things are done, at a certain step of the program code
- Many different ways to document the code **inline**
  - Use of the hashtag # symbol for a line comment
  - Use of the triple quotes for single or multiline string comments
  - In general, use of string constants for line comments

Inline comments: explaining the *how* part of the code

```
# let's start by defining an empty list
x = []

# fill the list with numbers from the user
n = int(input("How many numbers? "))
''' We should check if n is a number or not!
    As it is the program could crash if a wrong input is given'''

# get the user numbers, checking that they are actual numbers
while len(x) < n:
    nn = input("Next number: ")
    'check if it is a number or not'
    if nn.isnumeric():
        x.append(int(nn))

# compute the arithmetic average
s = sum(x)  # the sum first
avg = s / len(x) # then the division by the number of elements

# print out the result with some message
print("The average of the given", len(x), "numbers is:", avg)
```

Storing code in *modules* for flexible reusing and sharing

```
def get_average(l):

    if type(l) != list:
        return None

    if len(l) == 0:
        return 0

    avg = 0
    count = 0
    for i in range(len(l)):
        if (type(l[i]) != int and\
            type(l[i]) != float):
            continue
        else:
            avg += l[i]
            count += 1
    if count > 0:
        return avg / count
    else:
        return None
```

```
def get_median(l):

    if type(l) != list:
        return None

    if len(l) == 0:
        return 0

    for i in range(len(l)):
        if (type(l[i]) != int and type(l[i]) != float):
            return None

    l_sorted = sorted(l)
    median = l_sorted[ len(l_sorted)//2 ]
    return median
```

- Let's put them together and *store* in a **module**

→ **file** named `my_stats.py`

`import` (the entire module) and dot notation

- How do I use the functions in the module in another program? → `import` statement

```
import module_name
```

✓ `import my_stats`

➢ How do I invoke the functions inside `my_stats`?

➢ **Dot notation**! … the functions are *methods of the modules*

```
import my_stats

x = [2,7,5]
avg = my_stats.get_average(x)
mdn = my_stats.get_median(x)
print("Average:", avg, "Median:", mdn)
```

`import module_name` imports (loads) in the current program all the functions of the module

`import` only specific functions (no need dot notation)

- From large modules made by third parties, it might be heavy, time and memory taking, to load all functions, such as if only one or more functions are needed, they can be directly selected for import

```
from module_name import function_name
```

```
from module_name import function_1, function_2, function_3
```

```
from my_stats import get_average

x = [2,7,5]
avg = get_average(x)
print("Average:", avg)
```

- We don't need in this case the dot notation for invoking the function!
- The functions get actually *copied* into the current program module

`import` only specific functions (no need dot notation)

```
from my_stats import get_average

x = [2,7,5]
avg = get_average(x)
mdn = my_stats.get_median(x)
print("Average:", avg, "Median:", mdn)
```

Error!! `my_stats` isn't defined

Use aliases to refer to the module

- When in the code we need to refer many times to functions from a module, and maybe the name of the module is even long, it can be bothering / heavy to read to have such long names everywhere
- → Use an **alias** for the module name

```
import module_name as alias_name
```

```
import my_stats as st

x = [2,7,5]
avg = st.get_average(x)
mdn = st.get_median(x)
print("Average:", avg, "Median:", mdn)
```

Vs.

```
import my_stats

x = [2,7,5]
avg = my_stats.get_average(x)
mdn = my_stats.get_median(x)
print("Average:", avg, "Median:", mdn)
```

Importing all functions and getting rid of dots

- We can extend the `from` notation to import (i.e., copy) all functions from a module
- Since we *make a copy into the current module*, we don't need the dot notation anymore

```
from module_name import *
```

```
from my_stats import *

x = [2,7,5]
avg = get_average(x)
mdn = get_median(x)
print("Average:", avg, "Median:", mdn)
```

- **Not the best approach** since this *copy* might import variables, functions, etc. that might clash with variables and function names that we already have in the calling program

Importing modules from third parties: math

- Python offers a large number of specialized modules: you name it, it exists!
- Popular / useful modules (libraries):
  - `math:` mathematical functions https://docs.python.org/3/library/math.html

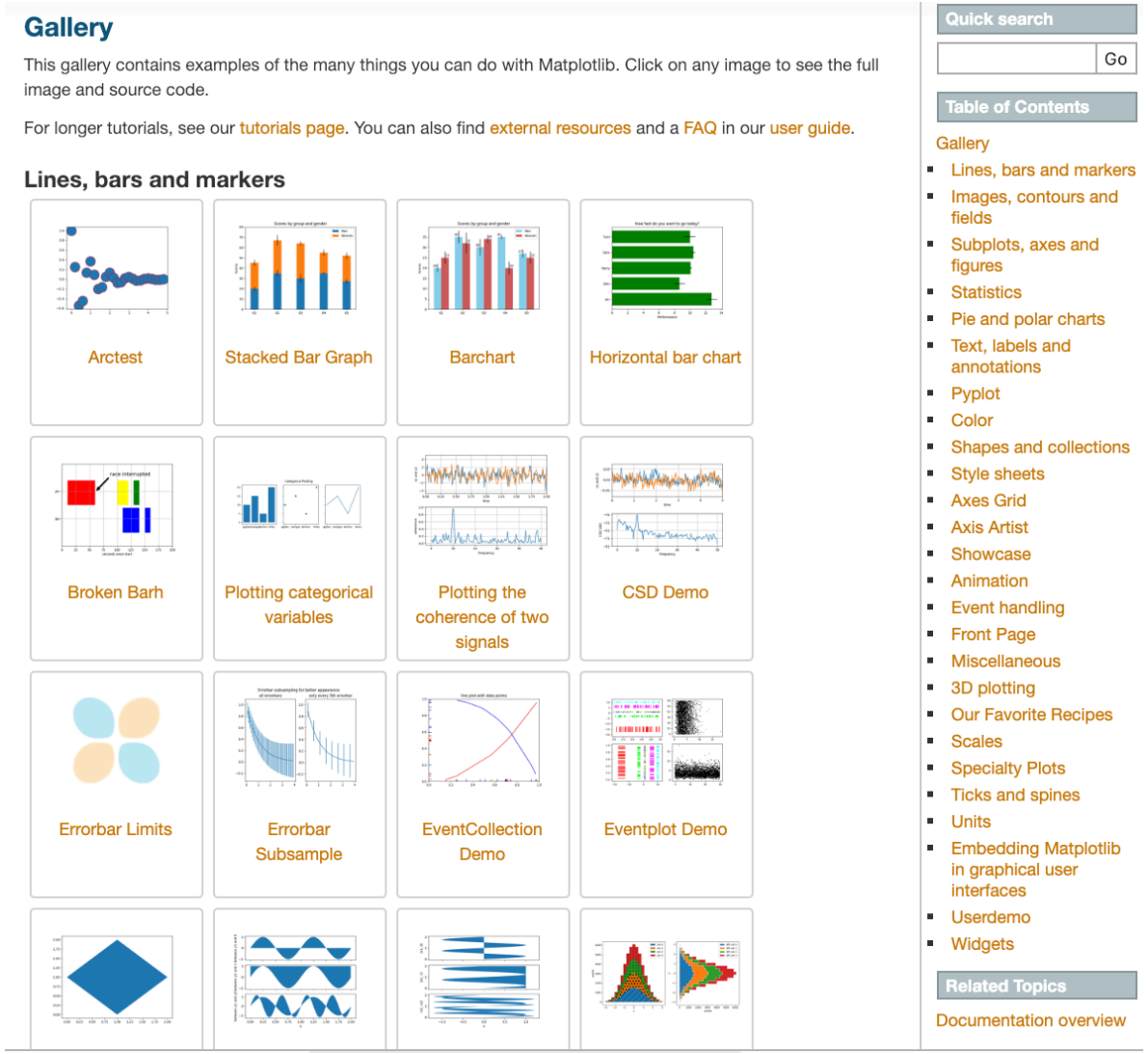
Function	Description
ceil(x)	Returns the smallest integer greater than or equal to x.
copysign(x, y)	Returns x with the sign of y
fabs(x)	Returns the absolute value of x
factorial(x)	Returns the factorial of x
floor(x)	Returns the largest integer less than or equal to x
fmod(x, y)	Returns the remainder when x is divided by y
frexp(x)	Returns the mantissa and exponent of x as the pair (m, e)
fsum(iterable)	Returns an accurate floating point sum of values in the iterable
isfinite(x)	Returns True if x is neither an infinity nor a NaN (Not a Number)
isinf(x)	Returns True if x is a positive or negative infinity
isnan(x)	Returns True if x is a NaN
ldexp(x, i)	Returns x * (2**i)
modf(x)	Returns the fractional and integer parts of x
trunc(x)	Returns the truncated integer value of x
exp(x)	Returns e**x
expm1(x)	Returns e**x - 1
log(x[, base])	Returns the logarithm of x to the base (defaults to e)
log1p(x)	Returns the natural logarithm of 1+x
log2(x)	Returns the base-2 logarithm of x
log10(x)	Returns the base-10 logarithm of x
pow(x, y)	Returns x raised to the power y
sqrt(x)	Returns the square root of x
acos(x)	Returns the arc cosine of x
asin(x)	Returns the arc sine of x
atan(x)	Returns the arc tangent of x
atan2(y, x)	Returns atan(y / x)
cos(x)	Returns the cosine of x
hypot(x, y)	Returns the Euclidean norm, sqrt(x*x + y*y)
sin(x)	Returns the sine of x
tan(x)	Returns the tangent of x
degrees(x)	Converts angle x from radians to degrees
radians(x)	Converts angle x from degrees to radians
acosh(x)	Returns the inverse hyperbolic cosine of x
asinh(x)	Returns the inverse hyperbolic sine of x
atanh(x)	Returns the inverse hyperbolic tangent of x
cosh(x)	Returns the hyperbolic cosine of x
sinh(x)	Returns the hyperbolic cosine of x
tanh(x)	Returns the hyperbolic tangent of x
erf(x)	Returns the error function at x
erfc(x)	Returns the complementary error function at x
gamma(x)	Returns the Gamma function at x
lgamma(x)	Returns the natural logarithm of the absolute value of the Gamma function at x
pi	Mathematical constant, the ratio of circumference of a circle to it's diameter (3.14159...)
e	mathematical constant e (2.71828...)

Importing modules from third parties: numpy

- `numpy`: numerical/scientific computing http://www.numpy.org/

Importing modules from third parties: matplotlib

- `matplotlib:` plotting library https://matplotlib.org/



Importing modules from third parties: random, CSV

- `random`: generating random numbers
- CSV: handle files in CSV format