LEARN CODING

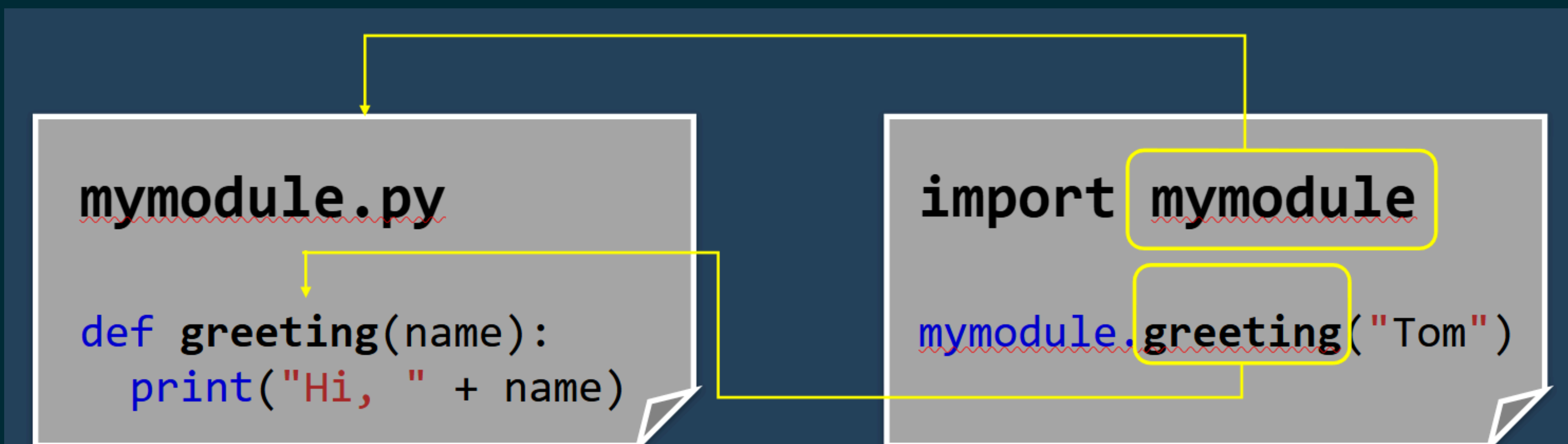
ale66

github.com/ale66/learn-coding

PYTHON MODULES

- functions allow to write a coherent, reusable code in isolation
- a *module* allows writing a coherent, reusable set of functions in isolation

- a module is a `.py` file that sits either locally or in designated locations controlled by the Python interpreter
- `pip` manages the designated locations and installs new modules
- an inclusion mechanism makes functions from a module file be listed in the *function space* of the program



PYTHON MODULES: MATH

```python
import math

mynum = 10

# This will print 3.16
print(math.sqrt(mynum)
```

Python » English 2.7.16 Documentation » The Python Standard Library » 9. Numeric and Mathematical Modules »

Table of Contents

- 9.2. math — Mathematical functions
  - 9.2.1. Number-theoretic and representation functions
  - 9.2.2. Power and logarithmic functions
  - 9.2.3. Trigonometric functions
  - 9.2.4. Angular conversion
  - 9.2.5. Hyperbolic functions
  - 9.2.6. Special functions
  - 9.2.7. Constants

Previous topic

9.1. numbers — Numeric abstract base classes

Next topic

9.3. cmath — Mathematical functions for complex numbers

This Page

Show Source

Quick search

Go

9.2. `math` — Mathematical functions

This module is always available. It provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for comple
numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numb
detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first pla

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.

9.2.1. Number-theoretic and representation functions

`math.ceil(x)`
Return the ceiling of *x* as a float, the smallest integer value greater than or equal to *x*.

`math.copysign(x, y)`
Return *x* with the sign of *y*. On a platform that supports signed zeros, `copysign(1.0, -0.0)` returns *-1.0*.

New in version 2.6.

`math.fabs(x)`
Return the absolute value of *x*.

`math.factorial(x)`
Return *x* factorial. Raises `ValueError` if *x* is not integral or is negative.

New in version 2.6.

`math.floor(x)`
Return the floor of *x* as a float, the largest integer value less than or equal to *x*.

`math.fmod(x, y)`
Return `fmod(x, y)`, as defined by the platform C library. Note that the Python expression `x % y` may not return the same result. The intent of th
precision) equal to `x - n*y` for some integer *n* such that the result has the same sign as *x* and magnitude less than `abs(y)`. Python's `x % y` returns a
float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be represented
function `fmod()` is generally preferred when working with floats, while Python's `x % y` is preferred when working with integers.

`math.frexp(x)`
Return the mantissa and exponent of *x* as the pair `(m, e)`. *m* is a float and *e* is an integer such that `x == m * 2**e` exactly. If *x* is zero, returns `(0.0,`
representation of a float in a portable way.

`math.fsum(iterable)`
Return an accurate floating point sum of values in the iterable. Avoids loss of precision by tracking multiple intermediate partial sums:

RANDOM NUMBERS

- Why we need random numbers?
  - Testing!
  - Generating test datasets
- Aside from obvious applications like gambling or creating unpredictable scenarios in a computer game, randomness is important for Cryptography
- Cryptography requires numbers that attackers *can't guess*

RANDOM NUMBERS (CONT.)

Are only pseudo-random

all languages have a dedicated module: crucial in several activities

Python implements Mersenne twisters; numpy has PCG64

```python
import random

# what does it print?
my_random_num = random.randint(1, 10)

print(my_random_num)

print(my_random_num)

print(my_random_num)
```

```python
import random

r1, r2, r3 = random.randint(1, 10), random.randint(1, 10), random.randint(1

print(r1)
print(r2)
print(r3)
```

```python
import random
alist = list()
for i in range(10):
  alist.append(random.randint(1, 10))
  print(alist)
[7, 3, 6, 6, 7, 5, 3, 3, 5, 5]
```

THE RANDOM MODULE

```python
from random import *

# Generate a pseudo-random number between 0 and 1
print(random())

# Pick a random number between 1 and 100
print(randint(1, 100))

# Shuffle the list items
items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(shuffle(items))

# Pick 4 random items from the list
print(sample(items, 4))
```

SEEDING RANDOM GENERATORS

- sometimes the pseudo-random sequence should be repeated for debuggin or validation
- The seed value is important in cyber-security to pseudo-randomly generate a strong secret encryption key.
- by re-using a custom seed value, we can initialize the strong pseudo-random number generator the way we want.

```python
import random

random.seed(1)

alist = list() # or []

for i in range(10):
  alist.append(random.randint(1, 10))

print(alist)

random.seed(2)

# empty out the list
alist = list()

for i in range(10):
  alist.append(random.randint(1, 10))
```

This always prints the same results!