Functions

- In Python, the definition of a function consists of four parts:
  - The function name;
  - The input argument: positional and keyword arguments;
  - The function body;
  - The return value;

```
def add(a, b):
    return a + b


def divide(dividend, divisor):
    if not divisor:
        raise ValueError("The divisor cannot be zero!")
    quotient = dividend // divisor
    remainder = dividend % divisor
    return quotient, remainder # these two numbers will be packed into a tuple
```

- In Python, only one function with a particular name can be defined in a scope. Defining another function with the same name overwrites the first function.

Default Arguments

- In Python, optional arguments must have default values and come after all the required arguments in the function definition. It is not necessary to pass optional arguments to a function.

```
    def make_greeting(title, name, surname, formal=True): # formal is an optional
 argument
        if formal:
            return "Hello, {} {}!".format(title, surname)
        return "Hello, {}!".format(name)


    print(make_greeting("Mr", "John", "Smith"))
    print(make_greeting("Mr", "John", "Smith", False))

    # the first two arguments are positional arguments and the last two are the ke
yword arguments
    print(make_greeting("Mr", "John", formal=False, surname="Smith"))
```

- When calling a function with positional and keyword arguments, the keyword argument always comes after the positional argument.
  - Positional arguments can also be passed by keyword.
  - Keyword arguments can be passed in any order as long as all of the required positional arguments are specified.
  - Keyword arguments are often used to define optional behaviors of a function.

Mutable Types and Default Arguments

Be careful when the default value is mutable because this default object is created at the function definition.

In [1]:

```
# notice that the default value for pets is mutable
def add_pet_to_list(pet, pets=[]):
    pets.append(pet)
    return pets

list_with_cat = add_pet_to_list("cat")
list_with_dog = add_pet_to_list("dog")
print(list_with_cat)
print(list_with_dog)
```

```
['cat', 'dog']
['cat', 'dog']
```

In [3]:

```
def add_pet_to_list(pet, pets=None):
    if pets is None:
        return [pet]
    else:
        pets.append(pet)
        return pets

list_with_cat = add_pet_to_list("cat")
list_with_dog = add_pet_to_list("dog")
print(list_with_cat)
print(list_with_dog)
```

```
['cat']
['dog']
```

*args and **kwargs

A function can accept a variable number of positional or keyword arguments.

A variable number of positional arguments can be passed in the following manner, where positional arguments are in a tuple and keyword arguments are in a dict.

A variable number of positional arguments

```
def print_args(*args):
    for arg in args:
        print(arg)
```

A variable number of keyword arguments

```
def print_kwargs(**kwargs):
    for kw, v in kwargs.items():
        print(kw, v)
```

A variable number of positional and keyword arguments

```
def print_args_kwargs(*args,**kwargs):
    for arg in args:
        print(arg)
    for kw, v in kwargs.items():
        print(kw, v)
```

- Use $*$ to unpack a sequence args to individual arguments for calling function f as f($*$args)
- Use $**$ to unpack a dict kwargs to individual keyword arguments for calling function f as f($**$kwargs)

```
a=[1,2,3]
f(*a)   # f(1,2,3)

d={'x': 10, 'y':20}
f(**d)  # f(x=10,y=20)
```

In [6]:

```
def f(x,y):
    return x + y

print(f(*[1,2]))
print(f(**{'x':10,'y':20}))
```

```
3
30
```

Scope

https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces (https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces)

In [7]:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

---

lambda functions

- The lambda keyword can be used to define anonymous, one-line functions inline in our code.
- A lambda function may only contain a single expression, and the result of evaluating this expression is implicitly returned from the function without using the return keyword.

```
a = lambda : -999
is the same as
def a():
    return -999


a = lambda x: x+1
is the same as
def a(x):
    return x + 1

a = lambda x,y : x<=y
is the same as
def a(x,y):
    return x<=y
```

The filter, map, and reduce Functions

filter(function, iterable) is used to select some elements in the iterable and its formula is equivalent to (item for item in iterable if function(item)).

In [10]:

```
a=[1,2,3,4,5,6,7,8]
p=filter(lambda x : x % 2 == 0,a)
for u in p:
    print(u)
```

```
2
4
6
8
```

map(function, iterable) is used to map every element in the iterable to another object or value, and its formula is equivalent to (function(item) for item in iterable).

In [9]:

```
a=[1,2,3,4]
p=map(lambda x : x + 2,a)
for u in p:
    print(u)
```

```
3
4
5
6
```

reduce(f, x), where f is a function and x is an iterable object, is used to perform some computation on x. The formula of reduce(f,x) is equivalent to

$$reduce(f,x) = \begin{cases} f(x[0],x[1]) & \text{if } len(x) == 2 \\ f(\ldots f(f(f(x[0],x[1]),x[2]),x[3]),\ldots),x[-1]) & \text{otherwise} \end{cases}$$

In [12]:

```
from functools import reduce
a=[1,2,3,4]
print(reduce(lambda x,y: x+y,a))
```

```
10
```

Decorators

- We may need to modify several functions in the same way for some reasons, for example, logging.

- In Python, we can write a function which modifies functions and call this function decorator. A decorator has a parameter which is the function to be modified and returns the modified function.

In [30]:

```
def logging(o_func):
    def n_function(*args, ** kwargs):
        print("{} called with positional arguments {} and keyword arguments {}.".format(o_f
        return o_func (*args,** kwargs)
    return n_function
```

In [31]:

```
# here is a function to decorate
def my_function1(message1,message2):
    print(message1,message2)
# decorate my_function
my_function1 = logging(my_function1)

my_function1('hello',message2='hi')
```

```
my_function1 called with positional arguments ('hello',) and keyword argumen
ts {'message2': 'hi'}.
hello hi
```

@logging defines my_function = logging(my_function)

In [32]:

```
# A shorthand syntax for applying decorators to functions
@logging
def my_function2(message1,message2):
    print(message1,message2)

my_function2('hello',message2='hi')
```

```
my_function2 called with positional arguments ('hello',) and keyword argumen
ts {'message2': 'hi'}.
hello hi
```

We can pass other parameters to decorators.

In [33]:

```
def another_logging(pre_msg):
    def n_func(o_func,*args, ** kwargs):
        print(pre_msg+"{} called with positional arguments {} and keyword arguments {}.".fo
        return o_func(*args,** kwargs)
    return lambda f: lambda *args, **kwargs : n_func(f, *args, **kwargs)
```

In [37]:

```
@another_logging("hi===>") # my_function = another_logging("hi")(my_function3)
def my_function3(message1,message2):
    print(message1,message2)

my_function3('hello',message2='hi')
```

```
hi===>my_function3 called with positional arguments ('hello',) and keyword a
rguments {'message2': 'hi'}.
hello hi
```

In [36]:

```
def tag(name):
    return lambda f : lambda *args, **kwargs: name + "." + f(*args,**kwargs)

@tag("decorator_A")#
@tag("decorator_B")#
def my_function4(x,y):
    return x+y

#my_function4 = tag("decorator_B")(my_function4)
#my_function4 = tag("decorator_A")(my_function4)

print(my_function4("hello","Python"))
```

```
decorator_A.decorator_B.helloPython
```

Generator Functions and the yield Statement

- The yield statement is used to create a generator. Whenever a new value is requested, the generator function will be resumed from the statement right after the yield statement previously executed.
- Once the generator function reaches the end of this function, a StopIteration exception arises.
- Consider a generator instead of a function returning a list.
  - Advantage: The working memory of a generator does not include all outputs!!!
  - Disadvantage: The iterator gotten from a generator cannot be reused.

In [5]:

```
def my_gen(n1, n2, divisor):
    for p in range(n1,n2):
        for x in divisor:
            if p % x == 0:
                break
        else:
            yield p
            print('m:',p) # my_gen will resume from here
    # Raise a StopIteration exception

for p in my_gen(3,16,[2,5,7,11]): # The for statement can handle StopIteration exception
    print('main_loop:',p)
```

```
main_loop: 3
m: 3
main_loop: 9
m: 9
main_loop: 13
m: 13
```

In [9]:

```
import random
def producer():
    while True:
        data = random.randint(0,100)
        print("produce {} units".format(data))
        yield data

def consumer():
    while True:
        data = yield
        print("consume {} units".format(data))

p = producer()
c = consumer()

# make consumer continue execution to a yield statement
next(c)

for i in range(10):
    c.send(next(p))
```

```
produce 60 units
consume 60 units
produce 70 units
consume 70 units
produce 47 units
consume 47 units
produce 54 units
consume 54 units
produce 47 units
consume 47 units
produce 64 units
consume 64 units
produce 91 units
consume 91 units
produce 28 units
consume 28 units
produce 27 units
consume 27 units
produce 96 units
consume 96 units
```

Built-in Functions

https://docs.python.org/3/library/functions.html (https://docs.python.org/3/library/functions.html)

Built-in Functions				
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

In [ ]: