Define a Function with def

To define a Python function, you type def, the function name, parentheses enclosing any input parameters to the function, and then finally, a colon (:). Function names have the same rules as variable names (they must start with a letter or _ and contain only letters, numbers, or _).

Let’s take things one step at a time, and first define and call a function that has no parameters.
Here’s the simplest Python function:

```
def do_nothing():
  pass
```

Call a Function with Parentheses

You call this function just by typing its name and parentheses. It works as advertised, doing nothing, but doing it very well:

```
do_nothing()
```

Now let’s define and call another function that has no parameters but prints a single word:

```
def make_a_sound():
  print('quack')

make_a_sound()
```

```
quack
```

```
When you called the make_a_sound() function, Python ran the code inside
its definition.
In this case, it printed a single word and returned to the main program.
Let’s try a function that has no parameters but returns a value:
```

```
def agree():
  return True
```

You can call this function and test its returned value by using if:

```
if agree():
  print('Splendid!')
else:
  print('That was unexpected.')
```

```
    Splendid!
```

```
def myfun():
  return 0

if myfun():
  print("India ")
else:
  print("Nothing")
```

```
    Nothing
```

Arguments and Parameters

```
At this point, it’s time to put something between those parentheses.
Let’s define the function echo() with one parameter called anything.
It uses the return statement to send the value of anything back to its
caller twice, with a space between:
```

```
def echo(anything): # here anything is a Parameter
  return anything + ' ' + anything
```

```
Now let’s call echo() with the string 'Rumplestiltskin':
```

```
```

```
echo('JISCE') # here JISCE is an Argument
```

```
     'JISCE JISCE'
```

```
The values you pass into the function when you call it are known as arguments.

When you call a function with arguments, the values of those arguments
are copied to their corresponding parameters inside the function.

Saying it another way: they’re called arguments outside of the func-
tion, but parameters inside.
```

```
In the previous example, the function echo() was called with the
argument string 'JISCE'.

This value was copied within echo() to the parameter anything,
and then returned (in this case doubled, with a space) to the caller.
```

```
These function examples were pretty basic.
Let’s write a function that takes an input argument and actually does
something with it.

We’ll adapt the code fragment that comments on a color.

Call it commentary and have it take an input string parameter called color.

Make it return the string description to its caller, which can decide
what to do with it:
```

```
def commentary(color):
    if color == 'red':
        return "It's a tomato."
    elif color == "green":
        return "It's a green pepper."
    elif color == 'bee purple':
        return "I don't know what it is, but only bees can see it."
    else:
        return "I've never heard of the color " + color + "."
```

```
Call the function commentary() with the string argument 'blue'.
```

```
comment = commentary('blue')
```

The function does the following:
- Assigns the value 'blue' to the function's internal color parameter
- Runs through the if-elif-else logic chain
- Returns a string

The caller then assigns the string to the variable comment.
What did we get back?

```
print(comment)
```

```
    I've never heard of the color blue.
```

A function can take any number of input arguments (including zero) of any type.
It can return any number of output results (also including zero) of any type.
If a function doesn't call return explicitly, the caller gets the result None.

```
print(do_nothing())
```

```
    None
```

None Is Useful

None is a special Python value that holds a place when there is nothing to say.
It is not the same as the boolean value False, although it looks false when evaluated as a boolean
Here's an example:

```
thing = None
if thing:
  print("It's some thing")
else:
  print("It's no thing")
```

```
    It's no thing
```

To distinguish None from a boolean False value, use Python’s is operator:

```
thing = None
if thing is None:
  print("It's nothing")
else:
  print("It's something")
```

```
It's nothing
```

```
a=10
b=10
a is b
```

```
True
```

This seems like a subtle distinction, but it’s important in Python.
You’ll need None to distinguish a missing value from an empty value.
Remember that zero-valued integers or floats, empty strings (''), lists ([]), tuples ((,)), dictionaries ({}), and sets(set()) are all False, but are not the same as None.

```
a=[]
a is True
```

```
False
```

Let’s write a quick function that prints whether its argument is None, True, or False:

```
def whatis(thing):
  if thing is None:
    print(thing, "is None")
  elif thing:
    print(thing, "is True")
  else:
    print(thing, "is False")
```

```
#Let’s run some sanity tests:
whatis(None)
```

```
None is None
```

```
whatis(True)
```

```
     True is True
```

```
whatis(False)
```

```
     False is False
```

```
#How about some real values?
whatis(0)
```

```
     0 is False
```

```
whatis(0.0)
```

```
     0.0 is False
```

```
whatis('')
```

```
      is False
```

```
whatis("")
```

```
      is False
```

```
whatis('''''')
```

```
      is False
```

```
whatis(())
```

```
     () is False
```

```
whatis([])
```

```
     [] is False
```

```
whatis({})
```

```
     {} is False
```

```
whatis(set())
```

```
     set() is False
```

```
whatis(0.00001)
```

```
     1e-05 is True
```

```
whatis([0])
```

```
     [0] is True
```

```
whatis([''])
```

```
     [''] is True
```

```
whatis(' ')
```

```
       is True
```

Positional Arguments

```
Python handles function arguments in a manner that’s very flexible, when compared to many language

The most familiar types of arguments are positional arguments,
whose values are copied to their corresponding parameters in order.

This function builds a dictionary from its positional input arguments
and returns it:
```

```
def menu(wine, entree, dessert):
  return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

```
menu('chardonnay', 'chicken', 'cake')
```

```
     {'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'cake'}
```

```
Although very common, a downside of positional arguments is that you
need to remember the meaning of each position.

If we forgot and called menu() with wine as the last argument instead
of the first, the meal would be very different:
```

```
menu('beef', 'bagel', 'bordeaux')
```

```
{'wine': 'beef', 'entree': 'bagel', 'dessert': 'bordeaux'}
```

Keyword Arguments

```
To avoid positional argument confusion, you can specify arguments by
the names of their corresponding parameters, even in a different order
from their definition in the function:
```

```
menu(entree='Mutton', dessert='bagel', wine='bordeaux')
```

```
{'wine': 'bordeaux', 'entree': 'Mutton', 'dessert': 'bagel'}
```

```
You can mix positional and keyword arguments.
Let’s specify the wine first, but use keyword arguments for the entree
and dessert:
```

```
menu('frontenac', dessert='flan', entree='fish')
```

```
{'wine': 'frontenac', 'entree': 'fish', 'dessert': 'flan'}
```

```
If you call a function with both positional and keyword arguments, the
positional arguments need to come first.
```

Specify Default Parameter Values

You can specify default values for parameters.
The default is used if the caller does not provide a corresponding argument.
This bland-sounding feature can actually be quite useful.
Using the previous example:

```
def menu(wine, entree, dessert='pudding'):
  return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

This time, try calling menu() without the dessert argument:

```
menu('chardonnay', 'chicken')
```

```
{'wine': 'chardonnay', 'entree': 'chicken', 'dessert': 'pudding'}
```

If you do provide an argument, it’s used instead of the default:

```
menu('dunkelfelder', 'duck', 'doughnut')
```

```
{'wine': 'dunkelfelder', 'entree': 'duck', 'dessert': 'doughnut'}
```

In the following test, the buggy() function is expected to run each
time with a fresh empty result list, add the arg argument to it, and
then print a single-item list.

However, there’s a bug: it’s empty only the first time it’s called.

The second time, result still has one item from the previous call:

```
def buggy(arg, result=[]):
  result.append(arg)
  print(result)
```

```
buggy('a')
```

```
['a']
```

```
buggy('b') #expect ['b']
```

```
['a', 'b']
```

It would have worked if it had been written like this:

```
def works(arg):
  result = []
  result.append(arg)
  return result
```

```
works('a')
```

```
['a']
```

```
works('b')
```

```
['b']
```

The fix is to pass in something else to indicate the first call:

```
def nonbuggy(arg, result=None):
  if result is None:
    result = []
  result.append(arg)
  print(result)
```

```
nonbuggy('a')
```

```
['a']
```

```
nonbuggy('b')
```

```
['b']
```

Explode/Gather Positional Arguments with *

```
If you've programmed in C or C++, you might assume that an asterisk (*)
in a Python program has something to do with a pointer.

Nope, Python doesn't have pointers.

When used inside the function with a parameter, an asterisk groups a
variable number of positional arguments into a single tuple of parameter values.
```

```
In the following example, args is the parameter tuple that resulted
from zero or more arguments that were passed to the function print_args():
```

```
def print_args(*args):
  print('Positional tuple:', args)
```

If you call the function with no arguments, you get nothing in *args:

```
print_args()
```

```
     Positional tuple: ()
```

Whatever you give it will be printed as the args tuple:

```
print_args(3, 2, 1, 'wait!', 'uh...')
```

```
     Positional tuple: (3, 2, 1, 'wait!', 'uh...')
```

```
This is useful for writing functions such as print() that accept a
variable number of arguments.


If your function has required positional arguments, as well, put them
first; *args goes at the end and grabs all the rest:
```

```
def ayan_priyo(*args):
  sum=0
  for arg in args:
    sum=sum+arg
  return sum

print(ayan_priyo(10,20))
print(ayan_priyo(10,20,30))
print(ayan_priyo(10,20,30,40))
print(ayan_priyo(10,20,30,40,50))
```

```
     30
     60
     100
     150
```

```
def print_more(required1, required2, *args):
  print('Need this one:', required1)
  print('Need this one too:', required2)
  print('All the rest:', args)
```

```
print_more('cap', 'gloves', 'scarf', 'monocle', 'mustache wax')
```

```
    Need this one: cap
    Need this one too: gloves
    All the rest: ('scarf', 'monocle', 'mustache wax')
```

When using *, you don’t need to call the tuple argument *args, but it’s a common idiom in Python.

It’s also common to use *args inside the function.

Summarizing:

- You can pass positional argument to a function, which will match them inside to positional param
- You can pass a tuple argument to a function, and inside it will be a tuple parameter. This is a simple case of the preceding one.
- You can pass positional arguments to a function, and gather them inside as the parameter *args, which resolves to the tuple args. This was described in this section.
- You can also “explode” a tuple argument called args to positional parameters *args inside the function, which will be regathered inside into the tuple parameter args.

- You can only use the * syntax in a function call or definition:

```
def print_args(*args):
  print('Positional tuple:', args)

print_args(2, 5, 7, 'x')
```

```
    Positional tuple: (2, 5, 7, 'x')
```

```
args = (2,5,7,'x')
print_args(args)
```

```
    Positional tuple: ((2, 5, 7, 'x'),)
```

```
print_args(*args)
```

```
    Positional tuple: (2, 5, 7, 'x')
```

```
args
```

```
    (2, 5, 7, 'x')
```

So:
• Outside the function, *args explodes the tuple args into comma-separated positional parameters.
• Inside the function, *args gathers all of the positional arguments into a single args tuple. You could use the names *params and params, but it’s common practice to use *args for both the outside argument and inside parameter.

Explode/Gather Keyword Arguments with **

You can use two asterisks (**) to group keyword arguments into a dictionary, where the argument names are the keys, and their values are the corresponding dictionary values.

The following example defines the function print_kwargs() to print its keyword arguments:

```
def print_kwargs(**kwargs):
  print('Keyword arguments:', kwargs)
```

Now try calling it with some keyword arguments:

```
print_kwargs()
```

```
     Keyword arguments: {}
```

```
print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')
```

```
     Keyword arguments: {'wine': 'merlot', 'entree': 'mutton', 'dessert': 'macaroon'}
```

Inside the function, kwargs is a dictionary parameter.

Argument order is:
• Required positional arguments
• Optional positional arguments (*args)
• Optional keyword arguments (**kwargs)

As with args, you don't need to call this keyword argument kwargs, but it's common usage.

The ** syntax is valid only in a function call or definition:

```
Summarizing:
• You can pass keyword arguments to a function, which will match them
inside to keyword parameters. This is what you've seen so far.
• You can pass a dictionary argument to a function, and inside it will
be dictionary parameters. This is a simple case of the preceding one.
• You can pass one or more keyword arguments (name=value) to a
function, and gather them inside as **kwargs, which resolves to the
dictionary parameter called kwargs. This was described in this section.
• Outside a function, **kwargs explodes a dictionary kwargs into
name=value arguments.
• Inside a function, **kwargs gathers name=value arguments into the
single dictionary parameter kwargs.
```

```
def ayan_priyaangshu(*args,**kwargs):
  for arg in args:
    print(arg)
  for k,v in kwargs.items():
    print(k,'\t',v)

ayan_priyaangshu(1,2,3,4,5,6,7,8,9,wine='merlot', entree='mutton', dessert='macaroon')
```

```
1
2
3
4
5
6
7
8
9
wine     merlot
entree   mutton
dessert          macaroon
```

Keyword-Only Arguments

It's possible to pass in a keyword argument that has the same name as a positional parameter, probably not resulting in what you want.

Python 3 lets you specify keyword-only arguments.

As the name says, they must be provided as name=value,not positionally as value.

The single * in the function definition means that the following parameters start and end must be provided as named arguments if we don't want their default values:

```
def print_data(data, *, start=0, end=100):
  for value in (data[start:end]):
    print(value)
```

```
data = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
print_data(data)
```

```
a
b
c
d
e
f
```

```
print_data(data, start=4)
```

```
e
f
```

```
print_data(data, end=2)
```

```
a
b
```

Mutable and Immutable Arguments

If an argument is mutable, its value can be changed from inside the function via its corresponding parameter:

```
outside = ['one', 'fine', 'day']
```

```
def mangle(arg):
  arg[1] = 'terrible!'
```

```
outside
```

```
['one', 'fine', 'day']
```

```
mangle(outside)
```

```
```

```
outside
```

```
['one', 'terrible!', 'day']
```

Docstrings

You can attach documentation to a
function definition by including a string at the beginning of the function body.
This is the function’s docstring:

```
def echo(anything):
  'echo returns its input argument'
  return anything
```

```
...
You can make a docstring quite long, and even add rich formatting if you want:
```

```
def print_if_true(thing, check):
  '''Prints the first argument if a second argument is true.The operation is:
    1. Check whether the *second* argument is true.
    2. If it is, print the *first* argument.
  '''
  if check:
    print(thing)
```

To print a function’s docstring, call the Python help() function.

Pass the function’s name to get a listing of arguments along with the nicely
formatted docstring:

```
help(echo)
```

```
Help on function echo in module __main__:

echo(anything)
    echo returns its input argument
```

If you want to see just the raw docstring, without the formatting:

```
print(echo.__doc__)   # __doc__
```

```
    echo returns its input argument
```

```
print(print_if_true.__doc__)
```

```
    Prints the first argument if a second argument is true.The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
```

```
That odd-looking __doc__ is the internal name of the docstring as a variable within the function.

Double underscores (aka dunder in Python-speak) are used in many
places to name Python internal variables, because programmers are unlikely to use them in their ow
```

Functions Are First-Class Citizens

The Python mantra, everything is an object.

This includes numbers,strings, tuples, lists, dictionaries—and functions, as well.

Functions are first-class citizens in Python.

You can assign them to variables, use them as arguments to other functions, and return them from functions.

This gives you the capability to do some things in Python that are difficult-to-impossible to carry out in many other languages.

```
To test this, let’s define a simple function called answer() that doesn’t have
any arguments; it just prints the number 42:
```

```
def answer():
  print(42)
```

If you run this function, you know what you’ll get:

```
answer()
```

```
42
```

```
Now let’s define another function named run_something.

It has one argument called func, a function to run.

Once inside, it just calls the function:
```

```
def run_something(func):
  func()
```

```
If we pass answer to run_something(), we’re using a function as data, just as
with anything else:
```

```
run_something(answer) # here answer is argument
```

```
42
```

```
Notice that you passed answer, not answer().

In Python, those parentheses mean call this function.

With no parentheses, Python just treats the function like any other object.

That’s because, like everything else in Python, it is an object:
```

```
type(run_something)
```

```
function
```

```
Let’s try running a function with arguments.

Define a function add_args() that prints the sum of its two numeric arguments, arg1 and arg2:
```

```
def add_args(arg1, arg2):
  print(arg1 + arg2)
```

And what is add_args()?

```
type(add_args)
```

```
function
```

```
At this point, let’s define a function called run_something_with_args() that takes three arguments

func :  The function to run
arg1 :  The first argument for func
arg2 :  The second argument for func
```

```
def run_something_with_args(func, arg1, arg2):
  func(arg1, arg2)
```

```
When you call run_something_with_args(), the function passed by the caller is
assigned to the func parameter, whereas arg1 and arg2 get the values that
follow in the argument list.

Then, running func(arg1, arg2) executes that function with those arguments
because the parentheses told Python to do so.

Let’s test it by passing the function name add_args and the arguments 5 and 9 to

run_something_with_args():
```

```
type(add_args)
```

```
function
```

```
run_something_with_args(add_args, 5, 9)
```

```
14
```

```
def add_args(arg1, arg2):
  print(arg1 + arg2)

def run_something_with_args(func, arg1, arg2):
  func(arg1, arg2)

run_something_with_args(add_args, 5, 9)
```

```
14
```

```
Within the function run_something_with_args(), the function name argument
add_args was assigned to the parameter func, 5 to arg1, and 9 to arg2.

This ended up running:
```

```
add_args(5, 9)
```

```
     14
```

```
You can combine this with the *args and **kwargs techniques.

Let’s define a test function that takes any number of positional arguments,
calculates their sum by using the sum() function, and then returns that sum:
```

```
def sum_args(*args):
  return sum(args)
```

```
I haven’t mentioned sum() before.

It’s a built-in Python function that calculates the
sum of the values in its iterable numeric (int or float) argument.

Let’s define the new function run_with_positional_args(), which takes a function
and any number of positional arguments to pass to it:
```

```
def run_with_positional_args(func, *args):
  return func(*args)
```

Now go ahead and call it:

```
run_with_positional_args(sum_args, 1, 2,3,4,5,6,7,8,9)
```

```
     45
```

```
def sum_args(*args):
  return sum(args)

def run_with_positional_args(func, *args):
  return func(*args)
```

```
run_with_positional_args(sum_args, 1, 2,3,4,5,6,7,8,9)
```

```
45
```

You can use functions as elements of lists, tuples, sets, and dictionaries.

Functions are immutable, so you can also use them as dictionary keys.

Inner Functions

```
You can define a function within another function:
```

```
def outer(a, b):  #4,7
  def inner(c, d): #4,7
    return c + d  #11
  return inner(a, b)  #4,7

outer(4, 7)
```

```
11
```

```
An inner function can be useful when performing some complex task more than
once within another function, to avoid loops or code duplication.

For a string example, this inner function adds some text to its argument:
```

```
def knights(saying):
  def inner(quote):
    return "We are the knights who say: '%s'" % quote
  return inner(saying)

knights('Ni!')
```

```
'We are the knights who say: 'Ni!''
```

Anonymous Functions: lambda

A Python lambda function is an anonymous function expressed as a single statement.

You can use it instead of a normal tiny function.

To illustrate it, let’s first make an example that uses normal functions.

To begin, let’s define the function edit_story(). Its arguments are the following:

- words—a list of words
- func—a function to apply to each word in words

```
def edit_story(words, func):
  for word in words:
    print(func(word))
```

Now we need a list of words and a function to apply to each word.

For the words,here’s a list of (hypothetical) sounds made by my cat if he (hypothetically) missed one of the stairs:

```
stairs = ['thud', 'meow', 'thud', 'hiss']
```

And for the function, this will capitalize each word and append an exclamation point, perfect for feline tabloid newspaper headlines:

```
def enliven(word): # give that prose more punch
  return word.capitalize() + '!'   #capitalizes the first letter

edit_story(stairs, enliven)
```

```
Thud!
Meow!
Thud!
Hiss!
```

Finally, we get to the lambda.

The enliven() function was so brief that we could replace it with a lambda:

```
edit_story(stairs, lambda word: word.capitalize() + '!')
```

```
     Thud!
     Meow!
     Thud!
     Hiss!
```

```
A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression.
```

```
lambda arguments : expression
```

```
x = lambda a : a + 10
print(x(5))
```

```
     15
```

```
x = lambda a, b : a * b
print(x(5, 6))
```

```
     30
```

```
def myfun(a,b):
  return a*b
y=myfun(5,6)
print(y)
```

```
     30
```

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

```
     13
```

```
def myfunc(n):#2
  return lambda a : a * n    #lambda a : a * 2   # lambda 11 : 11 * 2

mydoubler = myfunc(2)
print(mydoubler)
print(mydoubler(11))
```

```
<function myfunc.<locals>.<lambda> at 0x7f333567c5e0>
22
```

```
def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)
myquad= myfunc(4)
print(mydoubler(11))
print(mytripler(11))
print(myquad(11))
```

```
22
33
44
```

```
def myfunc(n):#2
  return lambda a,b : (a+b) * n    #lambda a : a * 2   # lambda 11,20 : (11+20) * 2

mydoubler = myfunc(2)
print(mydoubler(11,20))
```

```
<function myfunc.<locals>.<lambda> at 0x7f3335658670>
62
```

```
A lambda has zero or more comma-separated arguments, followed by a colon (:),
and then the definition of the function.

We’re giving this lambda one argument, word.

You don’t use parentheses with lambda as you would when calling a function
created with def.
```

yield Keyword

```
yield keyword is used to create a generator function.

A type of function that is memory efficient and can be used like an
iterator object.

In layman terms, the yield keyword will turn any expression that is
given with it into a generator object and return it to the caller.
```

```
Therefore, you must iterate over the generator object if you wish to
obtain the values stored there.
```

Example 1: Generator functions and yield Keyword in Python

```
Generator functions behave and look just like normal functions, but
with one defining characteristic.

Instead of returning data, Python generator functions use the yield
keyword.

Generators' main benefit is that they automatically create the
functions __iter__() and next ().

Generators offer a very tidy technique to produce data that is enormous
or limitless.
```

```
def fun_generator():
    yield "Hello world!!"
    yield "Machine Learning"
    yield 10

obj = fun_generator()

print(obj)
print(type(obj))

print(next(obj))
print(next(obj))
print(next(obj))
```

```
<generator object fun_generator at 0x7f334d2ed970>
<class 'generator'>
Hello world!!
Machine Learning
10
```

Example 2: Generating an Infinite Sequence

```
Here, we are generating an infinite sequence of numbers with yield,
yield returns the number and increments the num by + 1.

Note: Here we can observe that num+=1 is executed after yield but in
the case of a return, no execution takes place after the return keyword.
```

```
def inf_sequence():
    num = 0
    while True:
        yield num
        num += 1

for i in inf_sequence():
    print(i, end=" ")
```

Example 3: Demonstrating yield working with a list.

```
Here, we are extracting the even number from the list.
```

```
# generator to print even numbers
def print_even(test_list):  #[1, 4, 5, 6, 7,8,10]
    for i in test_list:
        if i % 2 == 0:
            yield i

# initializing list
test_list = [1, 4, 5, 6, 7,8,10]

# printing initial list
print("The original list is : " + str(test_list))

# printing even numbers
print("The even numbers in list are : ", end=" ")
for j in print_even(test_list):
    print(j, end=" ")
```

```
 The original list is : [1, 4, 5, 6, 7, 8, 10]
 The even numbers in list are :  4 6 8 10
```

Generators

A generator is a Python sequence creation object.

With it, you can iterate through potentially huge sequences without creating
and storing the entire sequence in memory at once.

Generators are often the source of data for iterators.

If you recall, we already used one of them, range(), in earlier code examples to generate a series

```
range(1,10)
```

```
range(1, 10)
```

```
sum(range(1, 101))
```

```
5050
```

Every time you iterate through a generator, it keeps track of where it was the
last time it was called and returns the next value.

This is different from a normal function,which has no memory of previous calls
and always starts at its first line with the same state.

Generator Functions

If you want to create a potentially large sequence, you can write a generator
function.

It’s a normal function, but it returns its value with a yield statement rather than return.

Let’s write our own version of range():

```
def my_range(first=0, last=10, step=1):
  number = first
  while number < last:
    yield number
    number += step
```

```
my_range
```

```
<function __main__.my_range(first=0, last=10, step=1)>
```

```
ranger = my_range(1, 5)
ranger
```

```
<generator object my_range at 0x7f334d210d60>
```

We can iterate over this generator object:

```
for x in ranger:
  print(x)
```

A generator can be run only once.

Lists, sets, strings, and dictionaries exist in memory, but a generator creates its values on the fly and hands them out one at a time through an iterator.

It doesn't remember them, so you can't restart or back up a generator.

If you try to iterate this generator again, you'll find that it's tapped out:

```
for try_again in ranger:
  print(try_again)
```

```
1
2
3
4
```

Generator Comprehensions

```
A generator comprehension looks like those, but is surrounded by parentheses
instead of square or curly brackets.

It's like a shorthand version of a generator function, doing the yield invisi-
bly, and also returns a generator object:
```

```
for i in zip(['a', 'b'], ['1', '2']):
  print(i)
```

```
('a', '1')
('b', '2')
```

```
genobj = (pair for pair in zip(['a', 'b'], ['1', '2']))
genobj
```

```
<generator object <genexpr> at 0x7f7c73077ad0>
```

```
for thing in genobj:
  print(thing)
```

```
('a', '1')
('b', '2')
```

Decorators

A decorator is a function that takes one function as input and returns another function.

Let’s dig into our bag of Python tricks and use the following:

- *args and **kwargs
- Inner functions
- Functions as arguments

The function document_it() defines a decorator that will do the following:

- Print the function’s name and the values of its arguments
- Run the function with the arguments
- Print the result
- Return the modified function for use

```
def document_it(func): #add_int
  def new_function(*args, **kwargs):
    print('Running function:', func.__name__)
    print('Positional arguments:', args)
    print('Keyword arguments:', kwargs)
    result = func(*args, **kwargs)
    print('Result:', result)
    return result
  return new_function
```

Whatever func you pass to document_it(), you get a new function that includes the extra statements that document_it() adds.

A decorator doesn’t actually have to run any code from func, but document_it() calls func partway through so that you get the results of func as well as all

the extras.

So, how do you use this? You can apply the decorator manually:

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

```
add_ints(3, 5)
```

```
8
```

```
cooler_add_ints = document_it(add_ints) # manual decorator assignment
```

```
cooler_add_ints(3, 5)
```

```
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

As an alternative to the manual decorator assignment we just looked at, you can add @decorator_name before the function that you want to decorate:

```
@document_it
def add_ints(a, b):
  return a + b

add_ints(3, 5)
```

```
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

You can have more than one decorator for a function.

Let’s write another decorator called square_it() that squares the result:

```
def square_it(func):
  def new_function(*args, **kwargs):
    result = func(*args, **kwargs)
    return result * result
  return new_function
```

```
The decorator that's used closest to the function (just above the def) runs first and then the one
Either order gives the same end result, but you can see how the intermediate steps change:
```

```
@document_it
@square_it
def add_ints(a, b):
  return a + b

add_ints(3, 5)
```

```
Running function: new_function
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 64
64
```

Let's try reversing the decorator order:

```
@square_it
@document_it
def add_ints(a, b):
  return a + b

add_ints(3, 5)
```

```
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
64
```

Exceptions

In some languages, errors are indicated by special function return values.

When things go south,Python uses exceptions: code that is executed when an associated error occurs.

It's good practice to add exception handling anywhere an exception might occur to let the user know what is happening.

Python prints an error message and some information about where the error occurred and then terminates the program, as demonstrated in the following snippet:

```
short_list = [1, 2, 3]
position = 5
short_list[position]
```

Handle Errors with try and except

Use try to wrap your code, and except to provide the error handling:

```
short_list = [1, 2, 3]
position = 5
try:
  print(short_list[position])
except:
  print('Need a position between 0 and', len(short_list)-1, ' but got',position)
```

```
Need a position between 0 and 2  but got 5
```

The code inside the try block is run. If there is an error, an exception is raised and the code inside the except block runs. If there are no errors, the except block is skipped.

```
short_list = [1, 2, 3]
while True:
  value = input('Position [q to quit]? ')
  if value == 'q':
    break
  try:
    position = int(value)
    print(short_list[position])
  except IndexError as err:
    print('Bad index:', position)
```