

**Basic Concepts** 





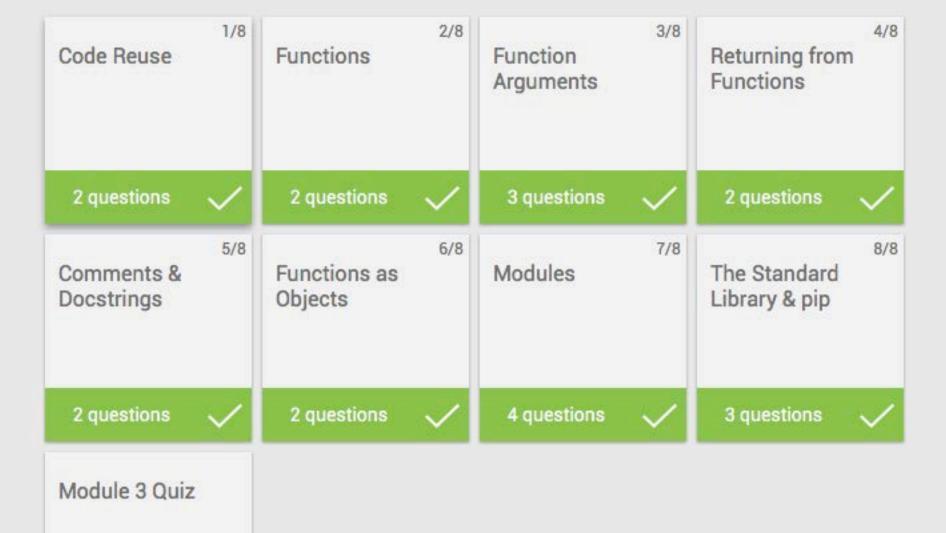




More Types



**Functional Programming** 



# **Reusing Code**

Code reuse is a very important part of programming in any language. Increasing code size makes it harder to maintain.

For a large programming project to be successful, it is essential to abide by the **Don't Repeat**Yourself, or DRY, principle. We've already looked at one way of doing this: by using loops. In this
module, we will explore two more: functions and modules.

Bad, repetitive code is said to abide by the WET principle, which stands for Write Everything Twice, or We Enjoy Typing.

105 COMMENTS





# Following the DRY principle makes the code:

0	bad and repetitive
0	easier to maintain
0	loop forever







## **Functions**

You've already used functions in previous lessons.

Any statement that consists of a word followed by information in parentheses is a function call. Here are some examples that you've already seen:

```
print("Hello world!")
range(2, 20)
str(12)
range(10, 20, 3)
```

The words in front of the parentheses are <u>function</u> names, and the comma-separated values inside the parentheses are <u>function</u> arguments.

**47 COMMENTS** 





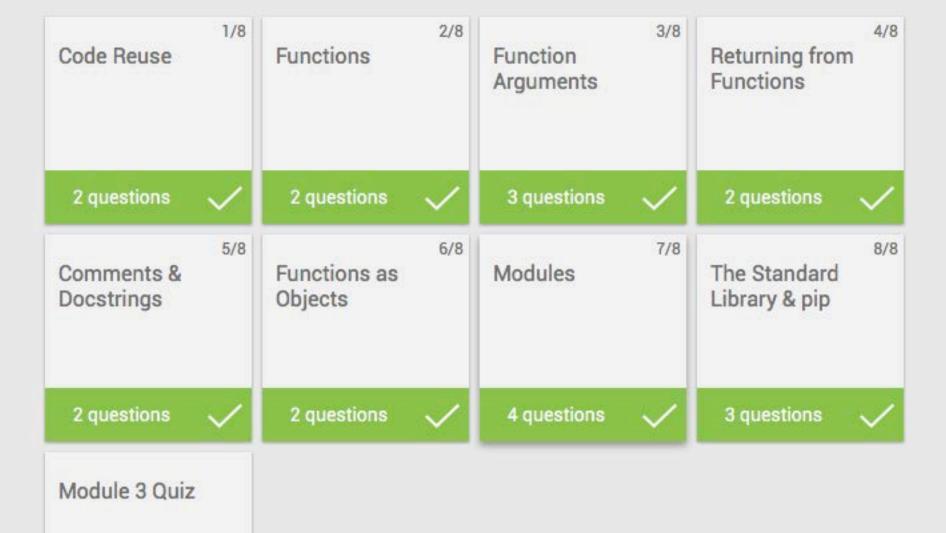
How many arguments are in this function call? range(0, 100, 5)











### **Functions**

In addition to using pre-defined functions, you can create your own functions by using the def statement.

Here is an example of a function named my\_func. It takes no arguments, and prints "spam" three times. It is defined, and then called. The statements in the function are executed only when the function is called.

```
def my_func():
    print("spam")
    print("spam")
    print("spam")

my_func()
```

Try It Yourself

#### Result:

```
>>>
spam
spam
spam
>>>
```

The code block within every function starts with a colon (:) and is indented.

Fill in the blanks to define a function named hello:

hello() print("Hi!")









## **Functions**

You must define functions before they are called, in the same way that you must assign variables before using them.

```
hello()

def hello():
 print("Hello world!")
```

**Try It Yourself** 

#### Result:

```
>>>
NameError: name 'hello' is not defined
>>>
```

**48 COMMENTS** 





# Rearrange the code to declare a function sayHi() and call it.

def sayHi():

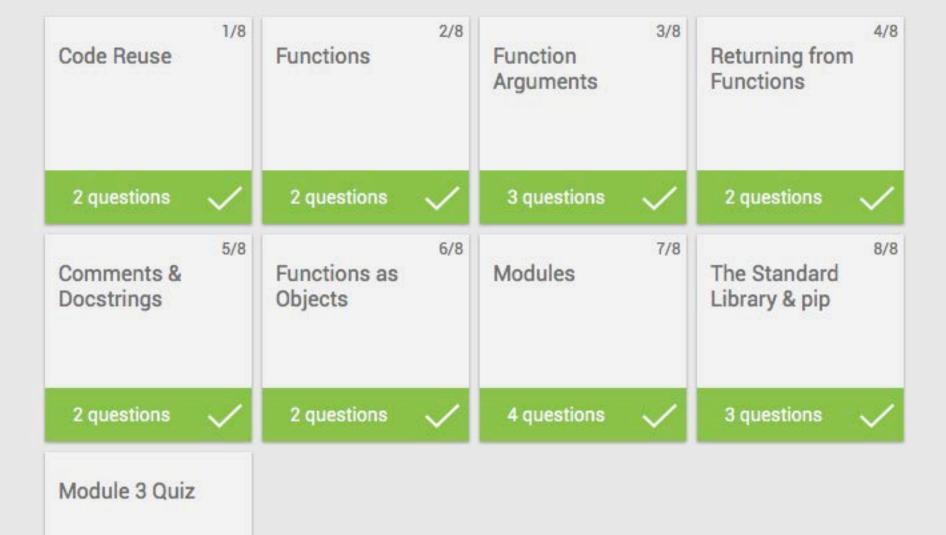
print("Hi!")

sayHi()









## **Arguments**

All the function definitions we've looked at so far have been functions of zero arguments, which are called with empty parentheses.

However, most functions take arguments.

The example below defines a function that takes one argument:

```
def print_with_exclamation(word):
    print(word + "!")

print_with_exclamation("spam")
print_with_exclamation("eggs")
print_with_exclamation("python")
```

Try It Yourself

### Result:

```
>>>
spam!
eggs!
python!
>>>
```

As you can see, the argument is defined inside the parentheses.

What is the result of this code? def print\_double(x): print(2 \* x)

print\_double(3)









# **Arguments**

You can also define functions with more than one argument; separate them with commas.

```
def print_sum_twice(x, y):
    print(x + y)
    print(x + y)

print_sum_twice(5, 8)
```

**Try It Yourself** 

### Result:

```
>>>
13
13
>>>
```

**49 COMMENTS** 





Fill in the blanks to define a function that takes two arguments and prints their multiplication.

```
__ print_mult(x, y) _
print(x * _)
```









## Arguments

Function arguments can be used as variables inside the function definition. However, they cannot be referenced outside of the function's definition. This also applies to other variables created inside a function.

```
def function(variable):
    variable += 1
    print(variable)

function(7)
print(variable)
```

**Try It Yourself** 

#### Result:

```
>>>
8

NameError: name '<u>variable</u>' is not defined
>>>
```

Technically, **parameters** are the variables in a <u>function</u> definition, and **arguments** are the values put into parameters when functions are called.

Fill in the blanks to define a function that prints "Yes", if its parameter is an even number, and "No" otherwise.

```
___ even(x):

if x%2 == 0:

___ ("Yes")

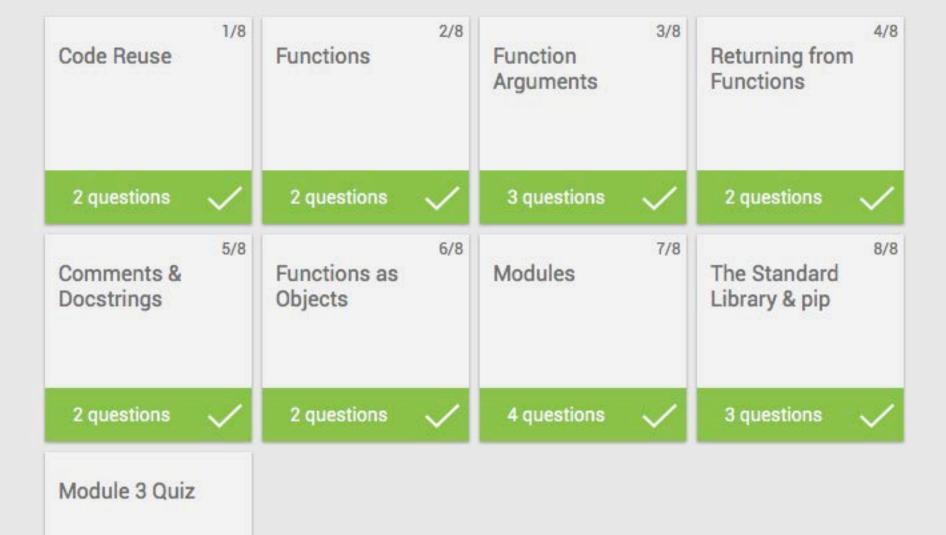
print("No")
```











# **Returning from Functions**

Certain functions, such as **int** or **str**, return a value that can be used later. To do this for your defined functions, you can use the **return** statement.

### For example:

```
def max(x, y):
    if x >= y:
        return x
    else:
        return y

print(max(4, 7))
z = max(8, 5)
print(z)
```

Try It Yourself

#### Result:

```
>>>
7
8
>>>
```

The return statement cannot be used outside of a function definition.

Fill in the blanks to define a function that compares the lengths of its arguments and returns the shortest one.

```
def shortest_string(x, y):
    if len(x) <= ___ (y):
        ___ x
    else:
    ___ y
```









# **Returning from Functions**

Once you return a value from a function, it immediately stops being executed. Any code after the return statement will never happen.

For example:

```
def add_numbers(x, y):
  total = x + y
  return total
  print("This won't be printed")
print(add_numbers(4, 5))
```

**Try It Yourself** 

### Result:

```
>>>
9
>>>
```

112 COMMENTS





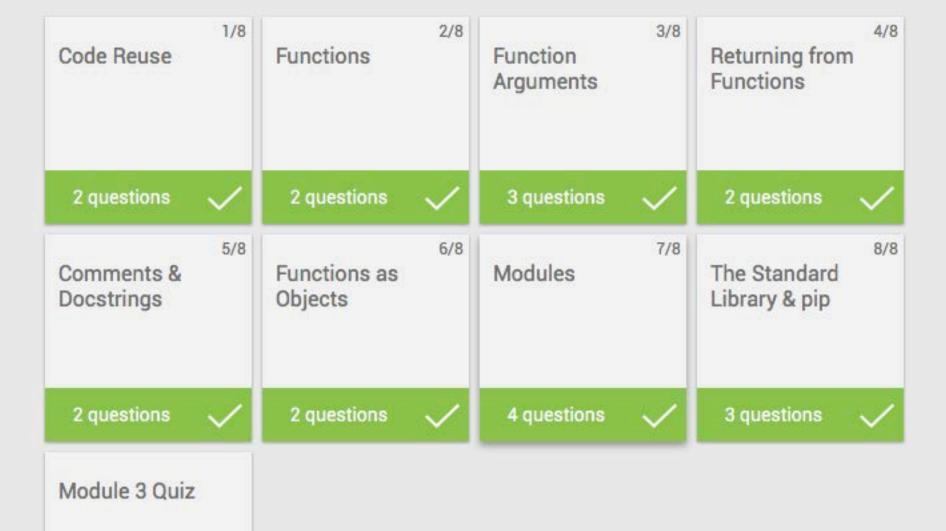
What is the highest number this function prints if called? def print\_numbers():
 print(1)
 print(2)
 return
 print(4)
 print(6)











### Comments

Comments are annotations to code used to make it easier to understand. They don't affect how code is run.

In Python, a comment is created by inserting an **octothorpe** (otherwise known as a number sign or hash symbol: #). All text after it on that line is ignored.

For example:

```
x = 365
y = 7
# this is a comment
print(x % y) # find the remainder
# print (x // y)
# another comment
```

Try It Yourself

### Result:

>>> 1 >>>

Python doesn't have general purpose multiline comments, as do programming languages such as C.

Fill in the blank to comment out the text:









# **Docstrings**

**Docstrings** (documentation strings) serve a similar purpose to comments, as they are designed to explain code. However, they are more specific and have a different syntax. They are created by putting a multiline string containing an explanation of the function below the function's first line.

```
def shout(word):

Print a word with an
exclamation mark following it.

"""
print(word + "!")
shout("spam")
```

Try It Yourself

#### Result:

```
>>>
spam!
>>>
```

Unlike conventional comments, <u>docstrings</u> are retained throughout the runtime of the program. This allows the programmer to inspect these comments at run time.

# Can a docstring contain multiple lines of text?

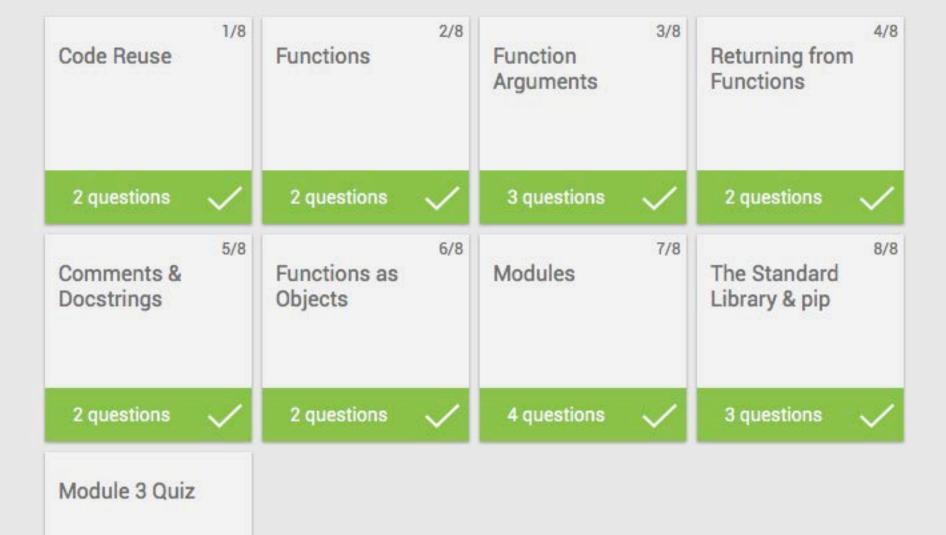
O No

O Yes









## **Functions**

Although they are created differently from normal variables, functions are just like any other kind of value.

They can be assigned and reassigned to variables, and later referenced by those names.

```
def multiply(x, y):
  return x * y

a = 4
b = 7
  operation = multiply
  print(operation(a, b))
```

Try It Yourself

#### Result:

```
>>>
28
>>>
```

The example above assigned the <u>function</u> multiply to a <u>variable</u> operation. Now, the name operation can also be used to call the <u>function</u>.

```
What is the output of this code?

def shout(word):

return word + "!"

speak = shout

output = speak("shout")

print(output)
```

- O shout!
- O word!
- O speak!







## **Functions**

Functions can also be used as arguments of other functions.

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

def do_twice(func, x, y):
    return func(func(x, y), func(x, y))

a = 5
b = 10

print(do_twice(add, a, b))
```

Try It Yourself

#### Result:

```
>>>
30
>>>
```

As you can see, the <u>function</u> **do\_twice** takes a <u>function</u> as its <u>argument</u> and calls it in its body.

Fill in the blanks to pass the function "square" as an argument to the function "test":

```
___ square(x):
return x * x

def test(func, x)
print(func(x))

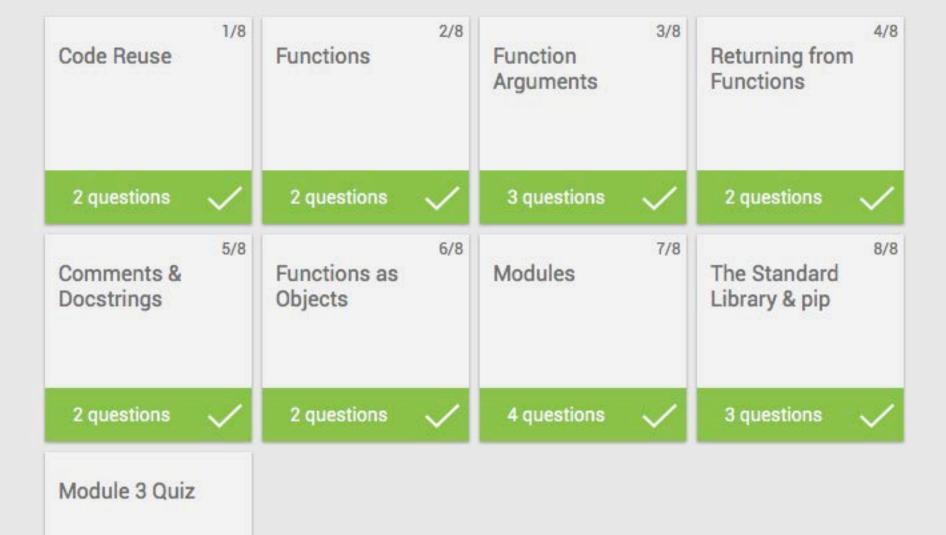
test(____, 42)
```











### Modules

Modules are pieces of code that other people have written to fulfill common tasks, such as generating random numbers, performing mathematical operations, etc.

The basic way to use a module is to add **import module\_name** at the top of your code, and then using **module\_name.var** to access functions and values with the name **var** in the module. For example, the following example uses the **random** module to generate random numbers:

```
import random

for i in range(5):
  value = random.randint(1, 6)
  print(value)
```

Try It Yourself

#### Result:

```
>>> 2
3
6
5
4
>>>>
```

The code uses the randint function defined in the random module to print 5 random

Which module is being used in this code? import math num = 10 print (math.sqrt(num))

0	math			
0	num			
0	sqrt			







There is another kind of **import** that can be used if you only need certain functions from a module. These take the form **from module\_name import var**, and then **var** can be used as if it were defined normally in your code.

For example, to import only the pi constant from the math module:

from math import pi
print(pi)

Try It Yourself

#### Result:

>>> 3.141592653589793 >>>

Use a comma separated list to import multiple objects. For example:

from math import pi, sqrt

\* imports all objects from a module. For example: from math import \*
This is generally discouraged, as it confuses variables in your code with variables in the external module.

Fill in the blanks to import only the sqrt and cos functions from the math module:

\_\_\_ math import \_\_\_ cos









Trying to import a module that isn't available causes an ImportError.

import some\_module

#### Result:

>>> ImportError: No module named 'some\_module' >>>





# What error is caused by importing an unknown module?

0	ImportError
0	UnknownModuleError
0	ModuleError







You can import a module or object under a different name using the **as** keyword. This is mainly used when a module or object has a long or confusing name.

#### For example:

from math import sqrt as square\_root print(square\_root(100))

**Try It Yourself** 

#### Result:

>>> 10.0 >>>





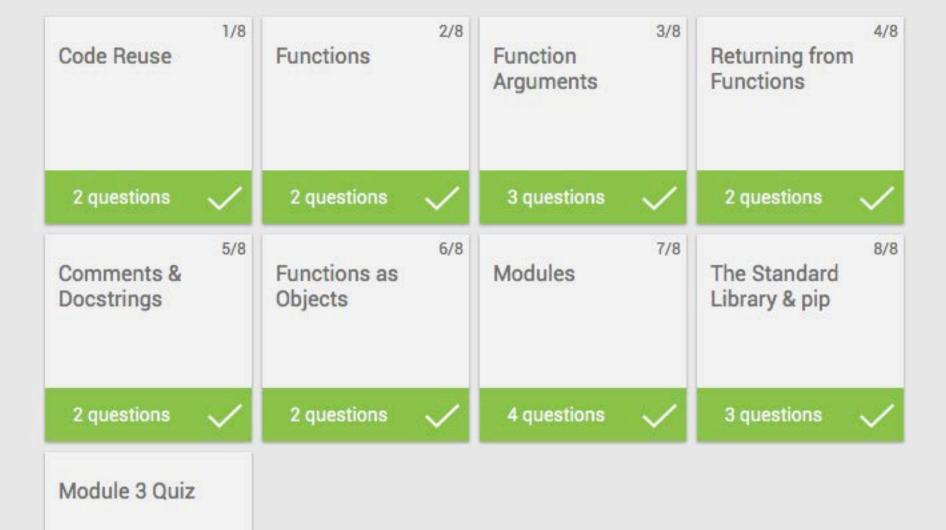
What is the output of this code? import math as m print(math.sqrt(25))

5An error occurs25









5 questions

There are three main types of modules in Python, those you write yourself, those you install from external sources, and those that are preinstalled with Python.

The last type is called the **standard library**, and contains many useful modules. Some of the standard library's useful modules include **string**, **re**, **datetime**, **math**, **random**, **os**, **multiprocessing**, **subprocess**, **socket**, **email**, **json**, **doctest**, **unittest**, **pdb**, **argparse** and **sys**.

Tasks that can be done by the standard library include string parsing, data serialization, testing, debugging and manipulating dates, emails, command line arguments, and much more!

Python's extensive standard library is one of its main strengths as a language.





Fill in the blanks to import the math module.

\_\_\_\_ math









## The Standard Library

Some of the modules in the standard library are written in Python, and some are written in C. Most are available on all platforms, but some are Windows or Unix specific.

We won't cover all of the modules in the standard library; there are simply too many. The complete documentation for the standard library is available online at www.python.org.





## What name is given to Python's preinstalled modules?

0	import
0	The Standard Library
0	Unix







Many third-party Python modules are stored on the **Python Package Index (PyPI)**. The best way to install these is using a program called **pip**. This comes installed by default with modern distributions of Python. If you don't have it, it is easy to install online. Once you have it, installing libraries from PyPI is easy. Look up the name of the library you want to install, go to the command line (for Windows it will be the Command Prompt), and enter **pip install library\_name**. Once you've done this, import the library and use it in your code.

Using **pip** is the standard way of installing libraries on most operating systems, but some libraries have prebuilt binaries for Windows. These are normal executable files that let you install libraries with a GUI the same way you would install other programs.

It's important to enter pip commands at the command line, not the Python interpreter.





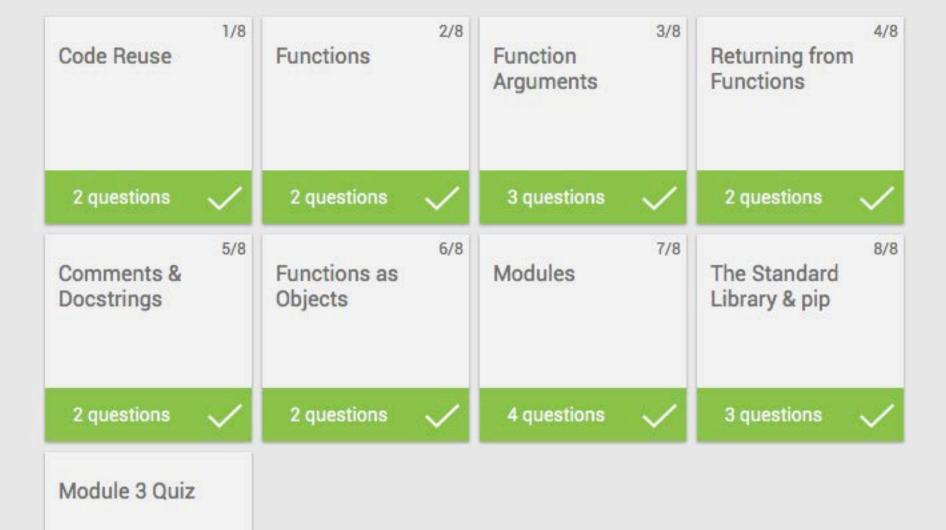
## What does PyPI stand for?

- O Python Package Index
- O Python Package Installer
- O Python Project Index









5 questions

Fill in the blanks to define a function that takes two numbers as arguments and returns the smaller one.

```
min(x, y):

if x<=y
return x
else:
y
```

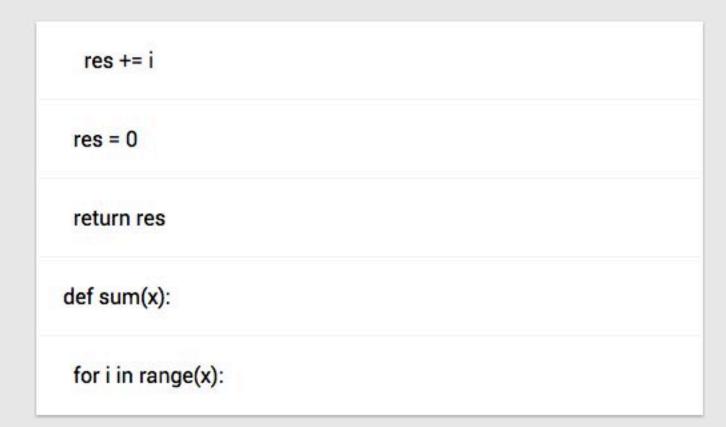








Rearrange the code to define a function that calculates the sum of all numbers from 0 to its argument.









# How would you refer to the randint function if it was imported like this? from random import randint as rnd\_int

0	randint
0	rnd_int
0	random.rnd_int







```
What is the highest number output by this code?

def print_nums(x):
  for i in range(x):
    print(i)
    return
print_nums(10)
```









```
What is the output of this code?

def func(x):
  res = 0
  for i in range(x):
    res += i
  return res

print(func(4))
```







