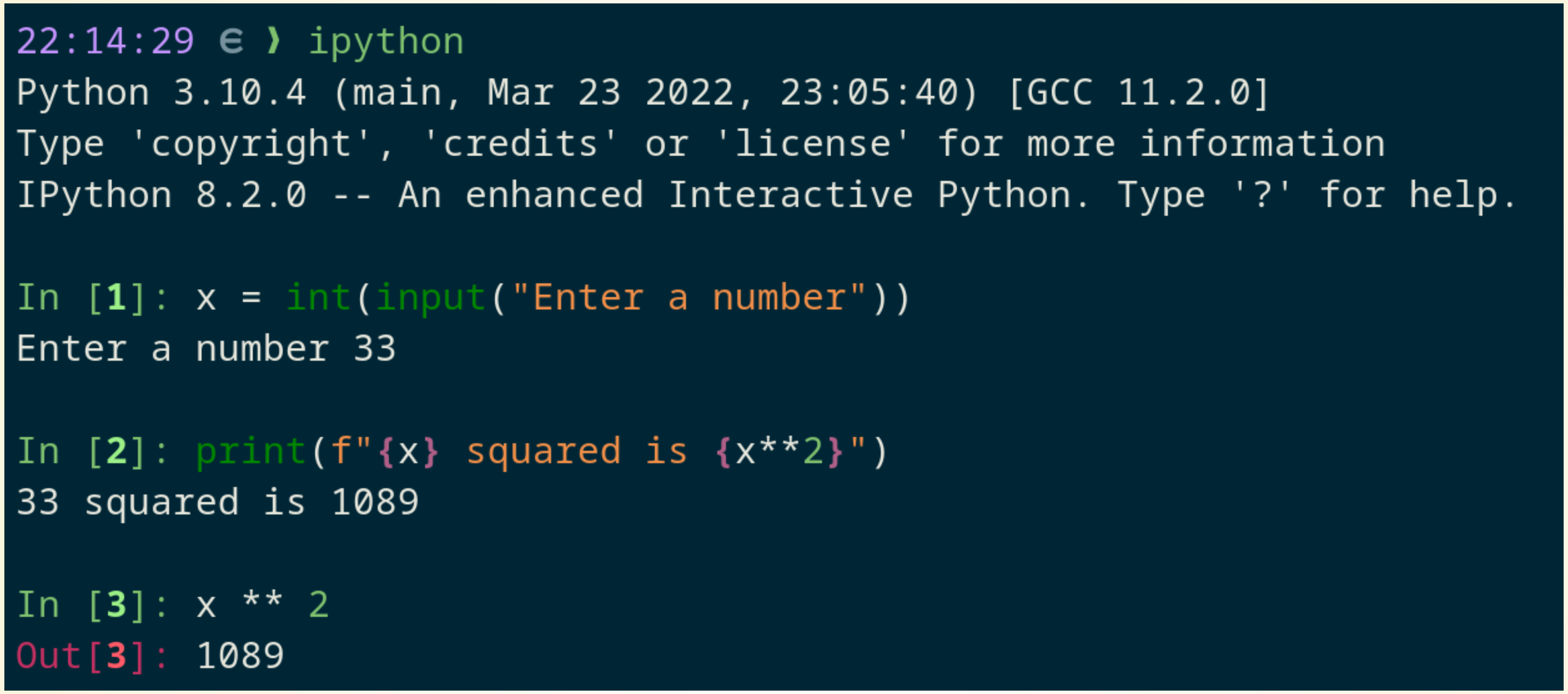
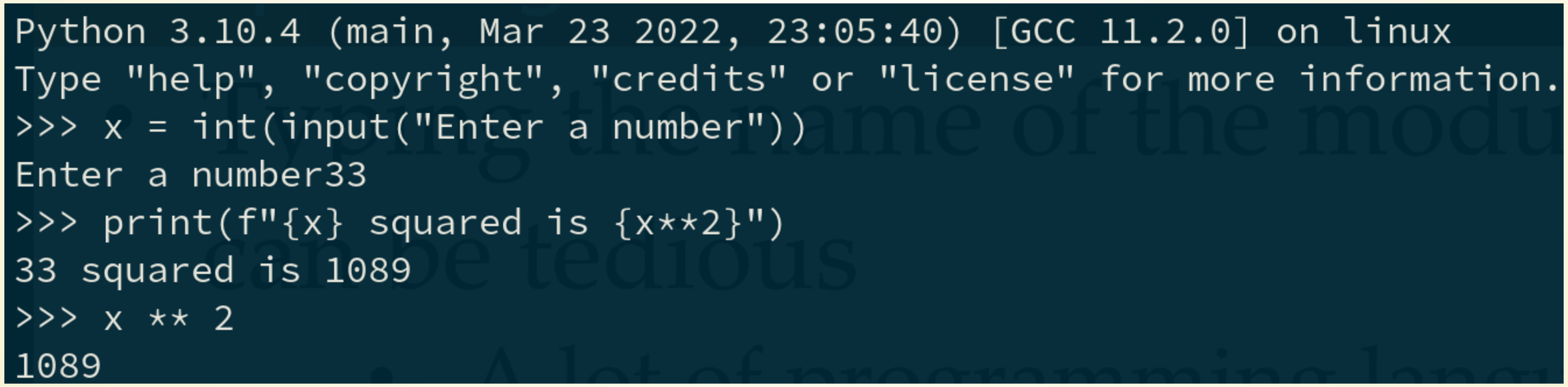
Functions

adapted from material by Mike Scott and Bill Young at the University of Texas at Austin

Note: IPython



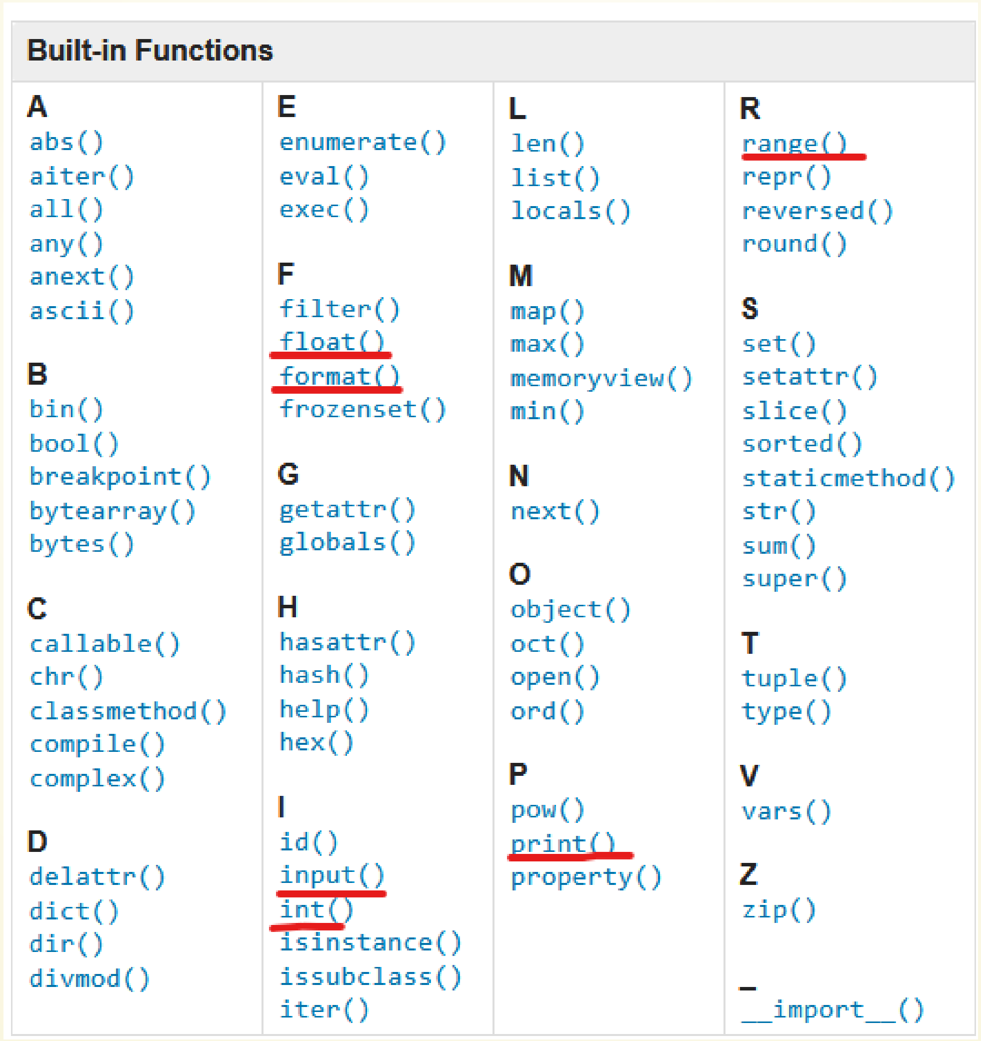
```
22:14:29 ∈ ❯ ipython
Python 3.10.4 (main, Mar 23 2022, 23:05:40) [GCC 11.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.2.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: x = int(input("Enter a number"))
Enter a number 33

In [2]: print(f"{x} squared is {x**2}")
33 squared is 1089

In [3]: x ** 2
Out[3]: 1089
```



```
Python 3.10.4 (main, Mar 23 2022, 23:05:40) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = int(input("Enter a number"))
Enter a number33
>>> print(f"{x} squared is {x**2}")
33 squared is 1089
>>> x ** 2
1089
```

Sometimes, we want to do the same thing over and over again, maybe with slight variations.

Functions allow us to do this. We have already used several built-in functions in Python.



Built-in Functions

A
abs()
aiter()
all()
any()
anext()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

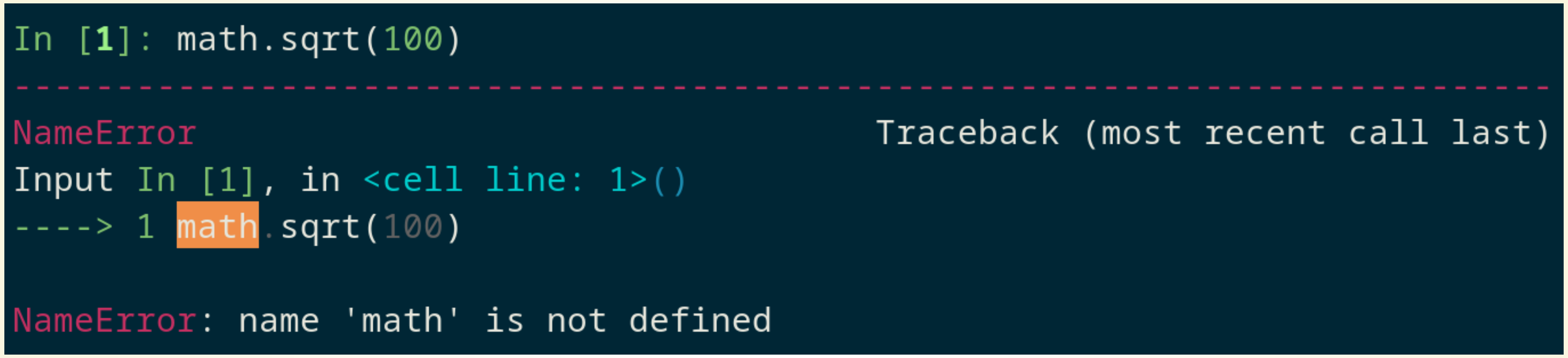
In addition, the python standard library comes with lots of other *modules.*

Modules are Python code that contain related functions which we can reuse. When you downloaded Python, you also downloaded the standard modules (also referred to as the "standard library").

Most of these modules are beyond the scope of this class.

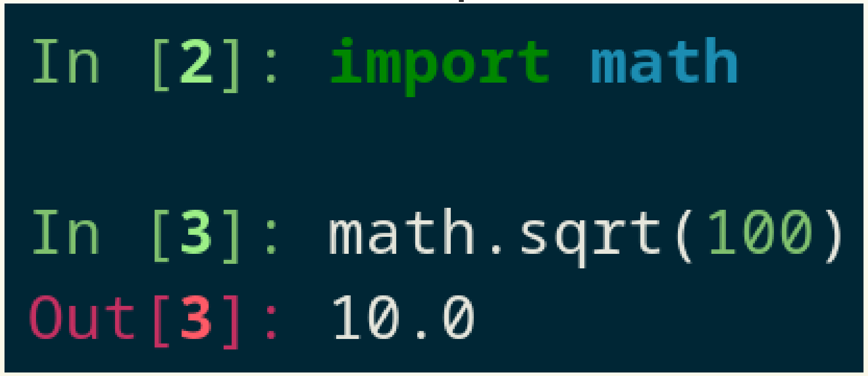
We will use the *math* module, which contains many common mathematical operations, and the *random* module, which contains functions for generating random numbers.

To use functions which are part of a module, we call the function with the name of the module, a period (pronounced "dot"), and the name of the function.



```
In [1]: math.sqrt(100)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [1], in <cell line: 1>()
----> 1 math.sqrt(100)

NameError: name 'math' is not defined
```

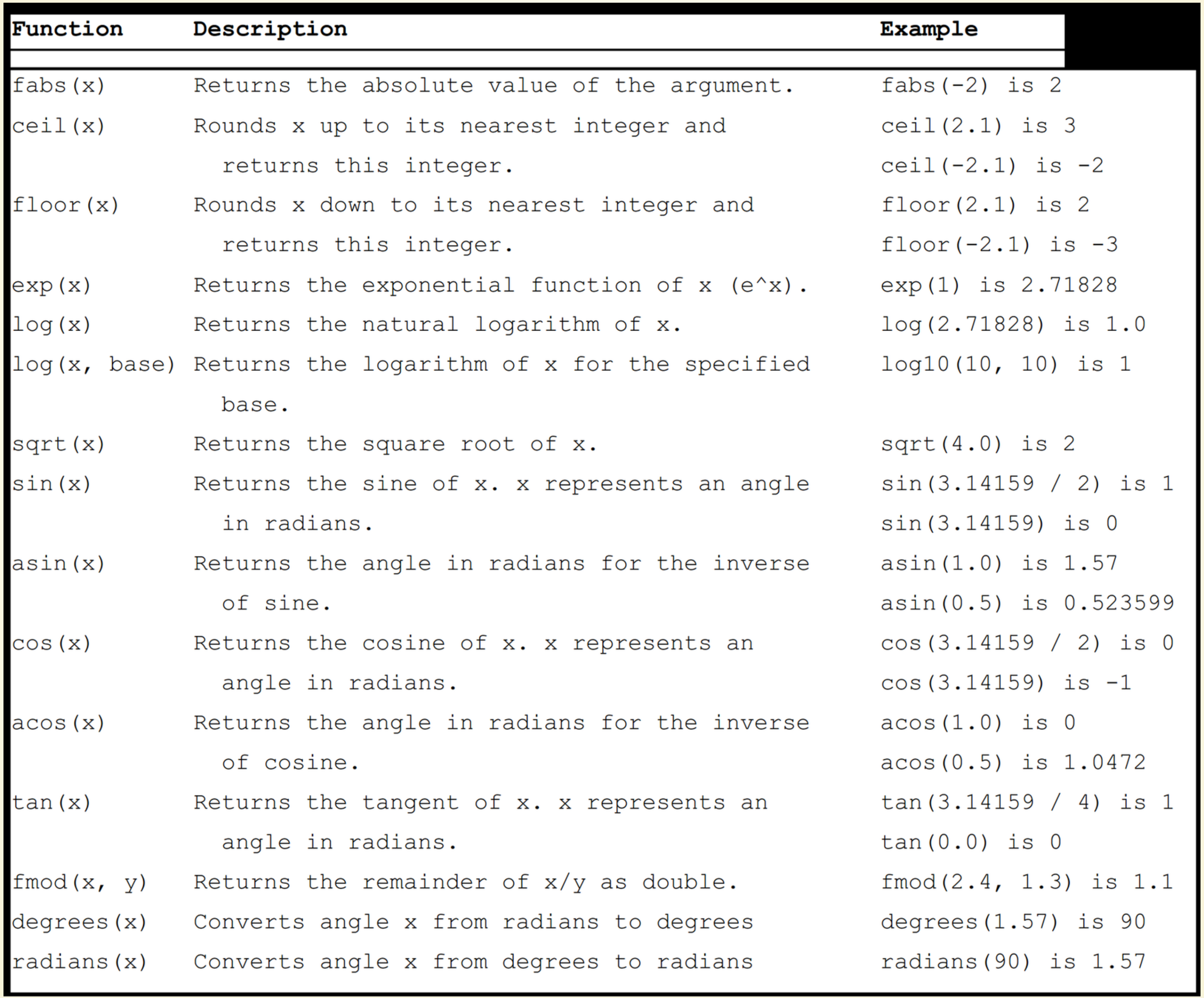
and we have to import the module:



```
In [2]: import math


In [3]: math.sqrt(100)
Out[3]: 10.0
```

If you're writing a file, imports should go at the very top of the file.

Lots of functions in math!



Function	Description	Example
fabs(x)	Returns the absolute value of the argument.	fabs(-2) is 2
ceil(x)	Rounds x up to its nearest integer and	ceil(2.1) is 3
	returns this integer.	ceil(-2.1) is -2
floor(x)	Rounds x down to its nearest integer and	floor(2.1) is 2
	returns this integer.	floor(-2.1) is -3
exp(x)	Returns the exponential function of x (e^x).	exp(1) is 2.71828
log(x)	Returns the natural logarithm of x.	log(2.71828) is 1.0
log(x, base)	Returns the logarithm of x for the specified	log10(10, 10) is 1
	base.	
sqrt(x)	Returns the square root of x.	sqrt(4.0) is 2
sin(x)	Returns the sine of x. x represents an angle	sin(3.14159 / 2) is 1
	in radians.	sin(3.14159) is 0
asin(x)	Returns the angle in radians for the inverse	asin(1.0) is 1.57
	of sine.	asin(0.5) is 0.523599
cos(x)	Returns the cosine of x. x represents an	cos(3.14159 / 2) is 0
	angle in radians.	cos(3.14159) is -1
acos(x)	Returns the angle in radians for the inverse	acos(1.0) is 0
	of cosine.	acos(0.5) is 1.0472
tan(x)	Returns the tangent of x. x represents an	tan(3.14159 / 4) is 1
	angle in radians.	tan(0.0) is 0
fmod(x, y)	Returns the remainder of x/y as double.	fmod(2.4, 1.3) is 1.1
degrees(x)	Converts angle x from radians to degrees	degrees(1.57) is 90
radians(x)	Converts angle x from degrees to radians	radians(90) is 1.57

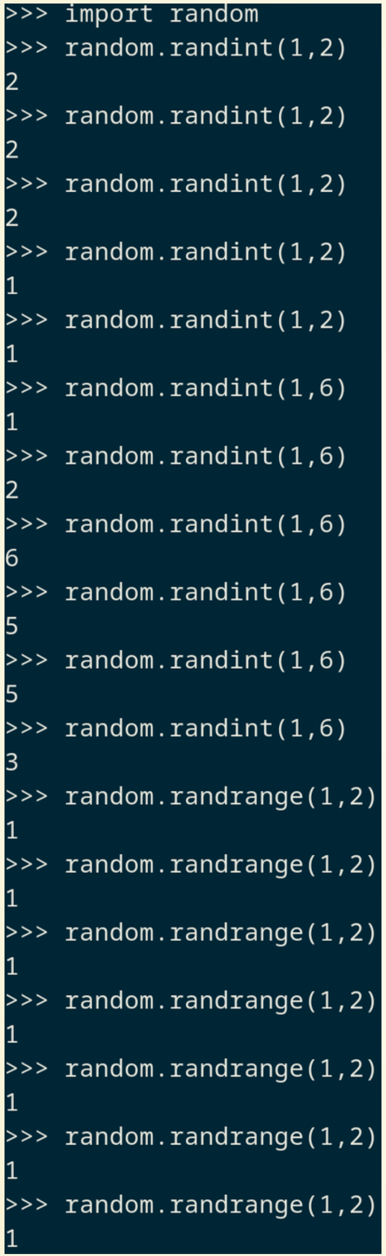
Random

There are several useful functions defined in random:

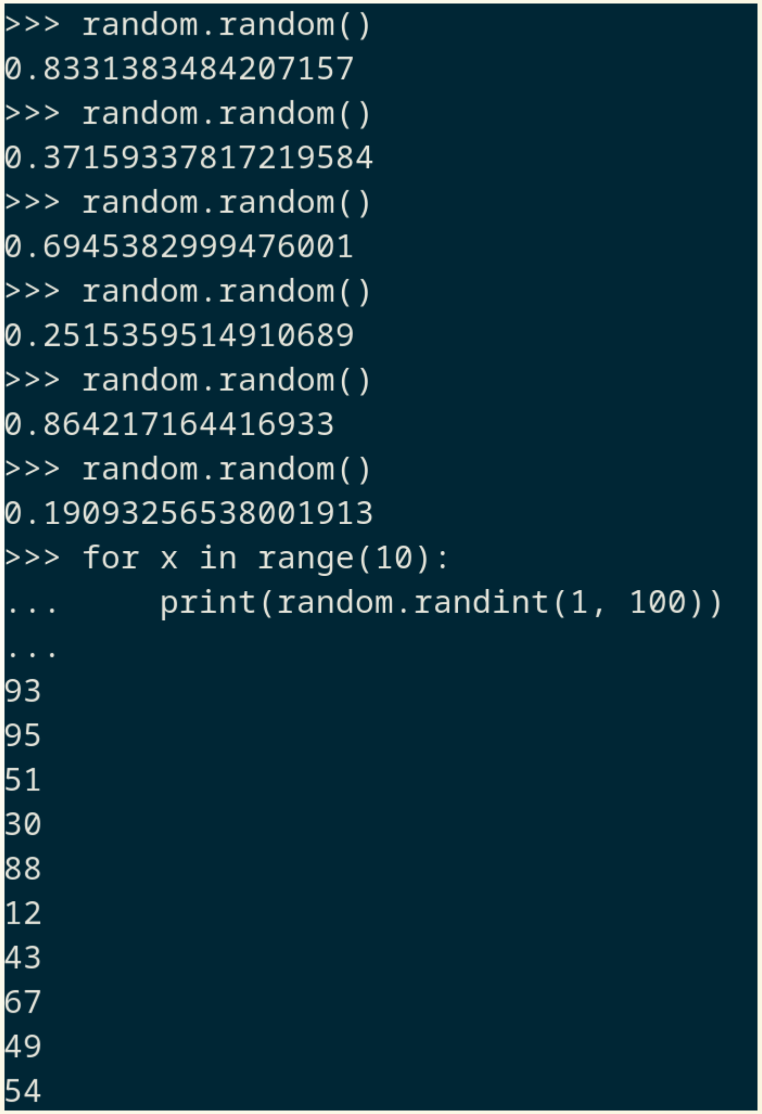
- randint(a, b): generate a random integer in $[a..b]$
- randrange(a,b): generate a random integer in $[a..b)$
- random(): generate a float in the range $(0..1)$

How should we simulate flipping a two-sided coin?

Examples with random



```
>>> import random
>>> random.randint(1,2)
2
>>> random.randint(1,2)
2
>>> random.randint(1,2)
2
>>> random.randint(1,2)
1
>>> random.randint(1,2)
1
>>> random.randint(1,6)
1
>>> random.randint(1,6)
2
>>> random.randint(1,6)
6
>>> random.randint(1,6)
5
>>> random.randint(1,6)
5
>>> random.randint(1,6)
3
>>> random.randrange(1,2)
1
>>> random.randrange(1,2)
1
>>> random.randrange(1,2)
1
>>> random.randrange(1,2)
1
>>> random.randrange(1,2)
1
>>> random.randrange(1,2)
1
>>> random.randrange(1,2)
1
```

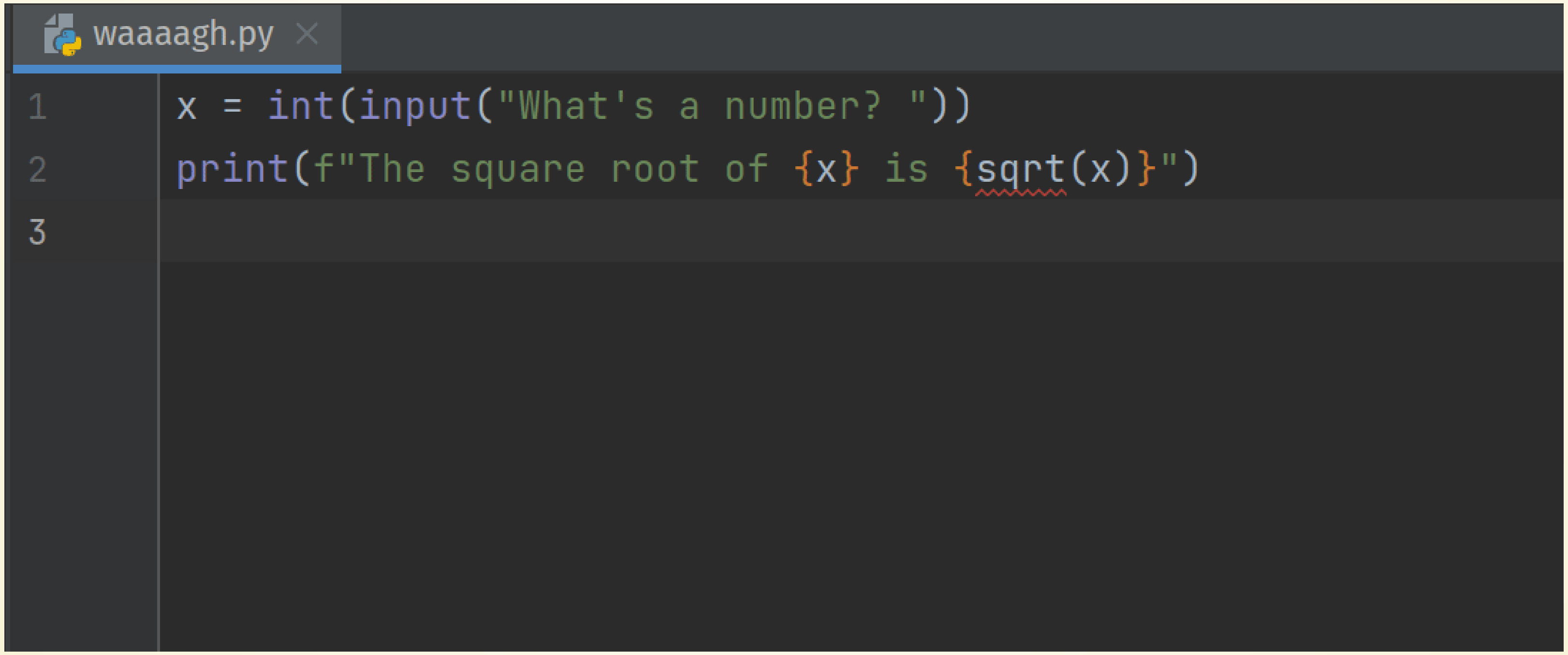


```
>>> random.random()
0.833138348420715‍7
>>> random.random()
0.3715933781721958‍4
>>> random.random()
0.694538299947600‍1
>>> random.random()
0.251535951491068‍9
>>> random.random()
0.86421716441693‍3
>>> random.random()
0.1909325653800191‍3
>>> for x in range(10):
...     print(random.randint(1, 100))
...
93
95
51
30
88
12
43
67
49
54
```

How to Import

Typing the name of the module every time can be annoying.

Most IDEs have features to reduce the amount of typing needed.

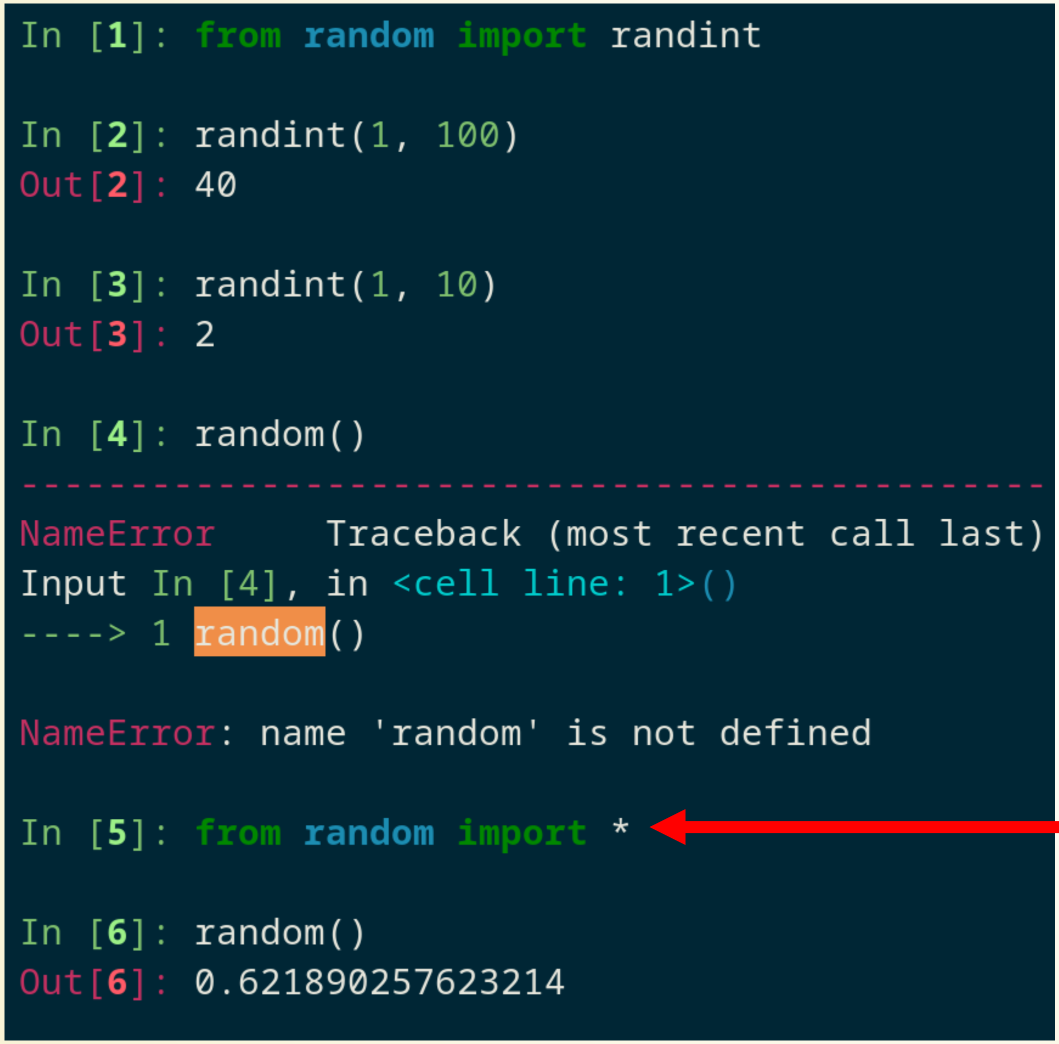


waaaagh.py

```python
x = int(input("What's a number? "))
print(f"The square root of {x} is {sqrt(x)}")

```

How to Import

We can also import specific functions from a module.



```
In [1]: from random import randint

In [2]: randint(1, 100)
Out[2]: 40

In [3]: randint(1, 10)
Out[3]: 2

In [4]: random()
---------------------------------------------------------------
NameError     Traceback (most recent call last)
Input In [4], in <cell line: 1>()
----> 1 random()

NameError: name 'random' is not defined

In [5]: from random import *

In [6]: random()
Out[6]: 0.621890257623214
```

import everything from random

Why not always import all?

Writing Functions

```python
def compute_max_of(num1, num2, num3):
  if num1 > num2 and num1 > num3:
    return num1
  elif num2 > num1 and num2 > num3:
    return num2
  else:
    return num3
```

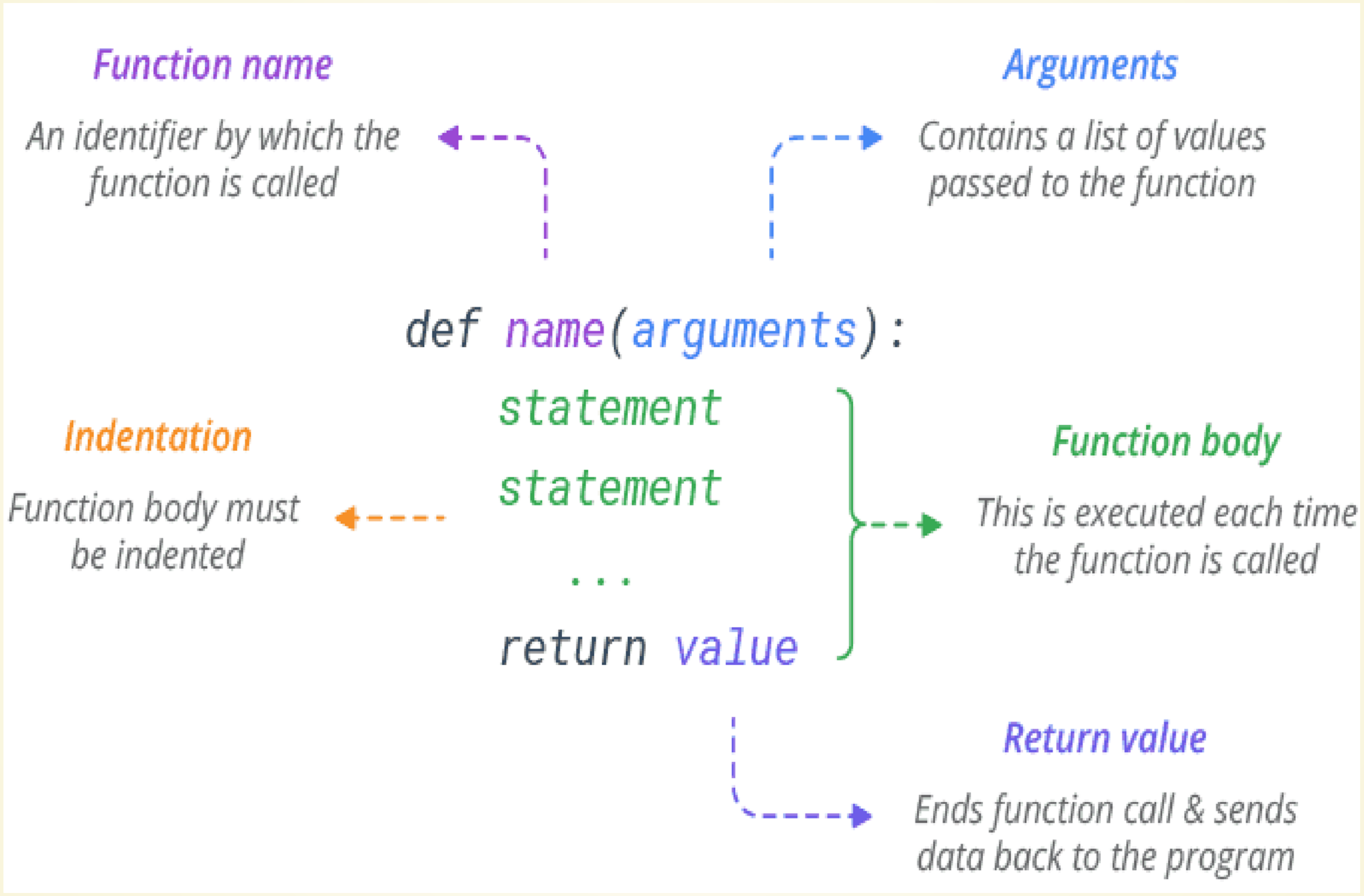
What happens when I call this?

```python
x = compute_max_of(7, 10, 13)
print(x)
```

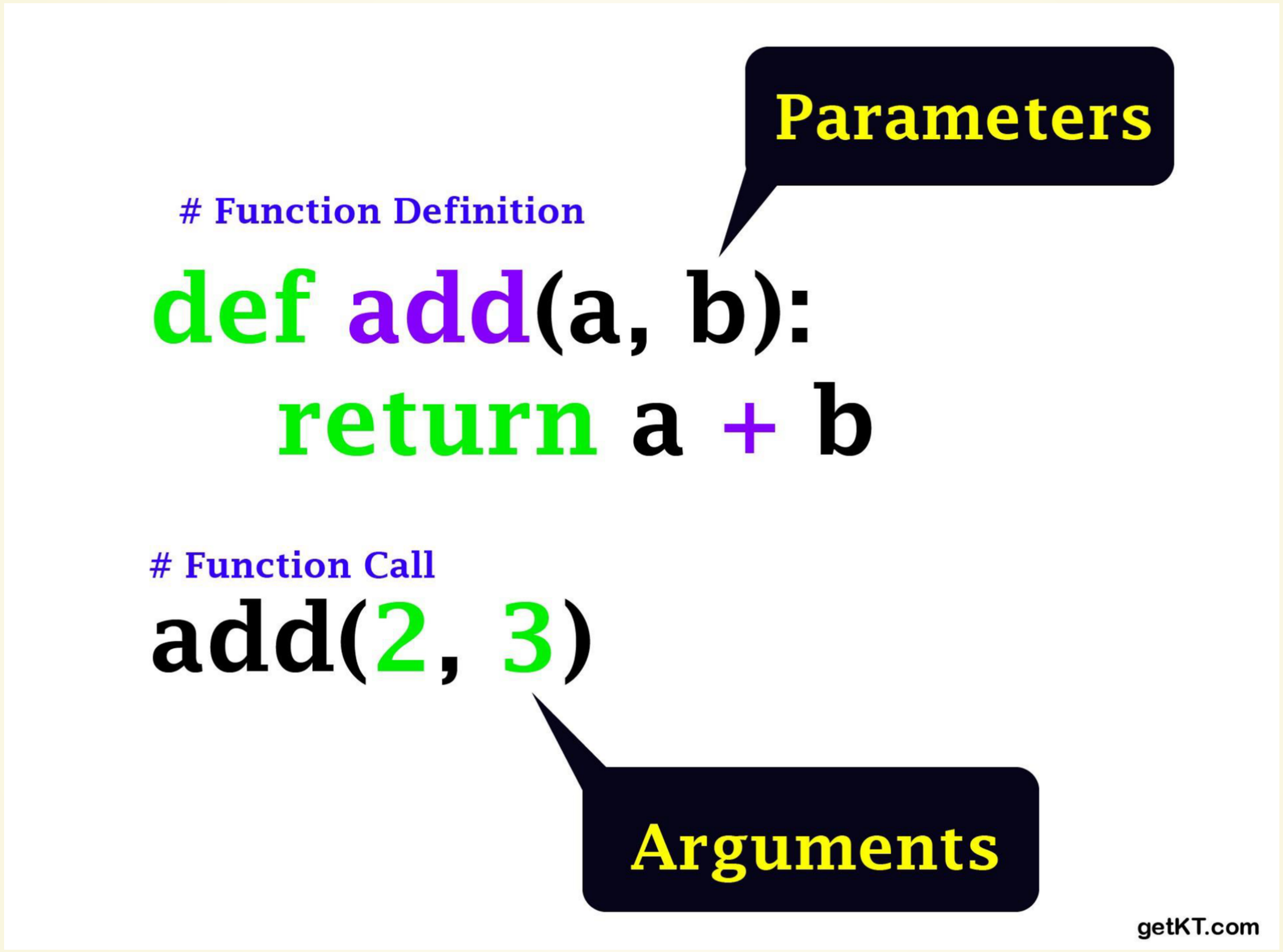
We've seen lots of system-defined functions. Now it's time to define our own.

```
def function_name( list of parameters ):
  body_statement_1
  body_statement_2
  ...
```

This defines a block of code that performs a specific task. It can reference any variables in the list of parameters. It may or may not return a value. These are also called arguments.


Function name
An identifier by which the function is called
Arguments
Contains a list of values passed to the function
Indentation
Function body must be indented
Function body
This is executed each time the function is called
Return value
Ends function call & sends data back to the program

```
def name(arguments):
    statement
    statement
      ...
    return value
```



```python
# Function Definition
def add(a, b):
    return a + b

# Function Call
add(2, 3)
```

Suppose you want to sum all the integers from 1 to n.

```python
# Returns the sum of values from 1 to n.
def sum_to_n(n):
  total = 0
  for i in range(1, n+1):
    total += i
  return total
```

Question: does this actually compute anything?

We still need to call the function! Otherwise, nothing actually gets run. A function acts like a "recipe" to do some computation, but writing the recipe down doesn't cause a cake to be baked.

```python
def main():
  print(sum_to_n(1))
  print(sum_to_n(1000))
```

Output:

```
1
500500
```

Observations

```python
def sum_to_n(n):
  ...
```

Here, n is a formal parameter. It is used as a placeholder for an actual parameter (e.g. 100, 1000) used in an actual call.

sum_to_int(n) returns an int value. This means that it can be used **anywhere** an int could be used.

```python
x = sum_to_n(30)
print(x)
print("Even" if sum_to_n(5) % 2 == 0 else "Odd")
for i in range(1, 30):
  print(i, sum_to_n(i))
```

Return

When a return statement is executed, you immediately return to the calling environment.

If you call return with a value, the caller gets that value.

If you call return without a value, Python implicitly returns None for you. **Do this carefully.**

Every function has an implicit return at the end.

```python
def print_x(x):
  print(x)

x = 10
y = print_x(x)
print(y)
```

Function In/Out

An important note: when we've been writing scripts so far, we've been using input() and print() to communicate with the user.

In general, for functions, you want to get inputs from arguments and pass outputs using return.

A common mistake:

```python
def double_num(x):
  y = 2 * x
  print(y)

def main():
  x = int(input("Enter a number to double: "))
  y = double_num(x)
  print(f"Double of {x} is {y}")
```

The Power of Abstraction

Once we've defined sum_to_n, we can use it without worrying about the definition.

We need to know what it does, <u>but we do not care about how it does it.</u>

This is called abstraction.

Used properly, it is incredibly, unbelievably powerful.

One day, we realize we can program sum_to_n much more efficiently using a method discovered by Gauss:

```python
# Retu[illegible] sum of values from 1 to n.
def [illegible]um_to_[illegible]:
  t[illegible]al = [illegible]
  f[illegible]r i [illegible] rang[illegible](1, n+1):
    [illegible]l += i
  ret[illegible]n tot[illegible]
```

```python
# Returns the sum of values from 1 to n.
def sum_to_n(n):
        return (n + 1) * n // 2
```

```python
def main():
  max_num = 1000000
  max_sum = sum_to_n(max_num)
  print(f"The sum of the first {max_num} numbers is {max_sum}")
```

We don't have to change any other code!! If we'd written the code in-line, we'd have to find all other instances.

More Examples

and docstrings

Docstrings

Python supports a special version of documentation for a function called a docstring. It goes right below the function definition at the same indentation as the body.

```python
def add_nums(a, b):
  """Adds a and b together"""
  return a + b
```

```python
def dummy_func(a, b):
  """If we need them to, docstrings can even be multiline!

     a: an input
     b: an input
     returns: the sum of a and b
  """
  return a + b
```

Other Examples

```python
def multiply_to_n(n):
    """Multiplies all values from 1 to n. This is
       also known as a factorial function"""
    result = 1
    for i in range(2, n+1):
      result *= result
```

This function...doesn't work.

Convert °C to °F, and °F to °C

```python
def fahrenheit_to_celsius(degrees_f):
    return 5 / 9 * (degrees_f - 32)

def celsius_to_fahrenheit(degrees_c):
    return 9 / 5 * degrees_c + 32
```

Write a program which prints a conversion table.

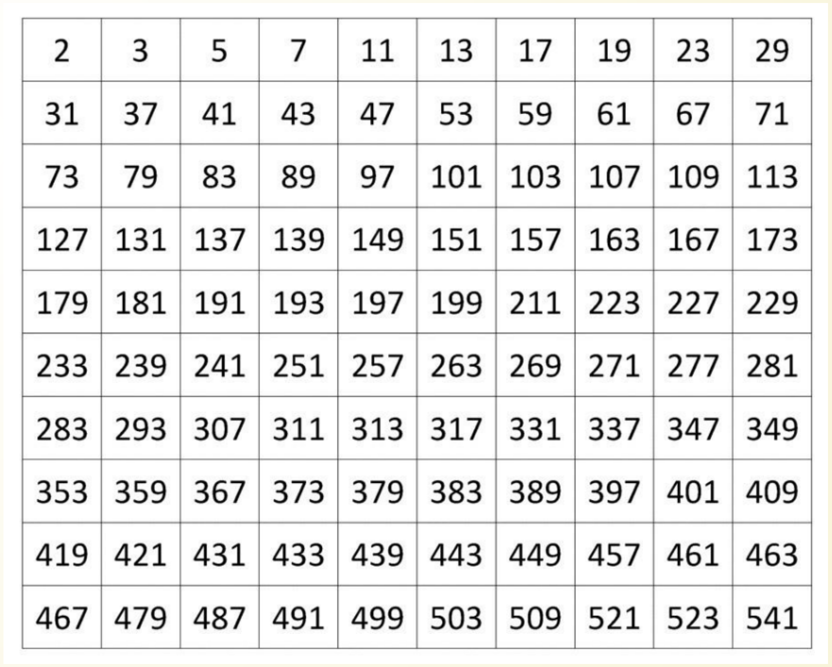
Convert °F in the range [-80, 130] every 10 °F.

Primes

Suppose you want to print out a table of the first 100 primes, 10 per line

You could do this without functions, but it would be a *mess*.

Much better idea: find and use functional abstraction. Find parts of the algorithm that can be coded separately and "packaged" as functions.



2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541

Here's some Python-like pseduocode to print 100 primes:

```
def print_100_primes():
  prime_count = 0
  num = 0
  while prime_count < 100:
    if (we already printed 10 on this line):
      go to a new line
    next_prime = (the next prime > num)
    print next_prime on the current line
    num = next_prime
    prime_count = 1
```

Note that most of this is straightforward python. The only new part is how to find the next prime. Let's make that a function.

Assuming we can get the next prime, write a function to print the first n primes, 10 per row.

Now that we have that, let's write a function to get the next prime.

Pseudocode:

```
def get_next_prime(num):
  if num < 2:
    return 2
  else:
    guess = num + 1
    while (guess is not prime)
      guess += 1
    return guess
```

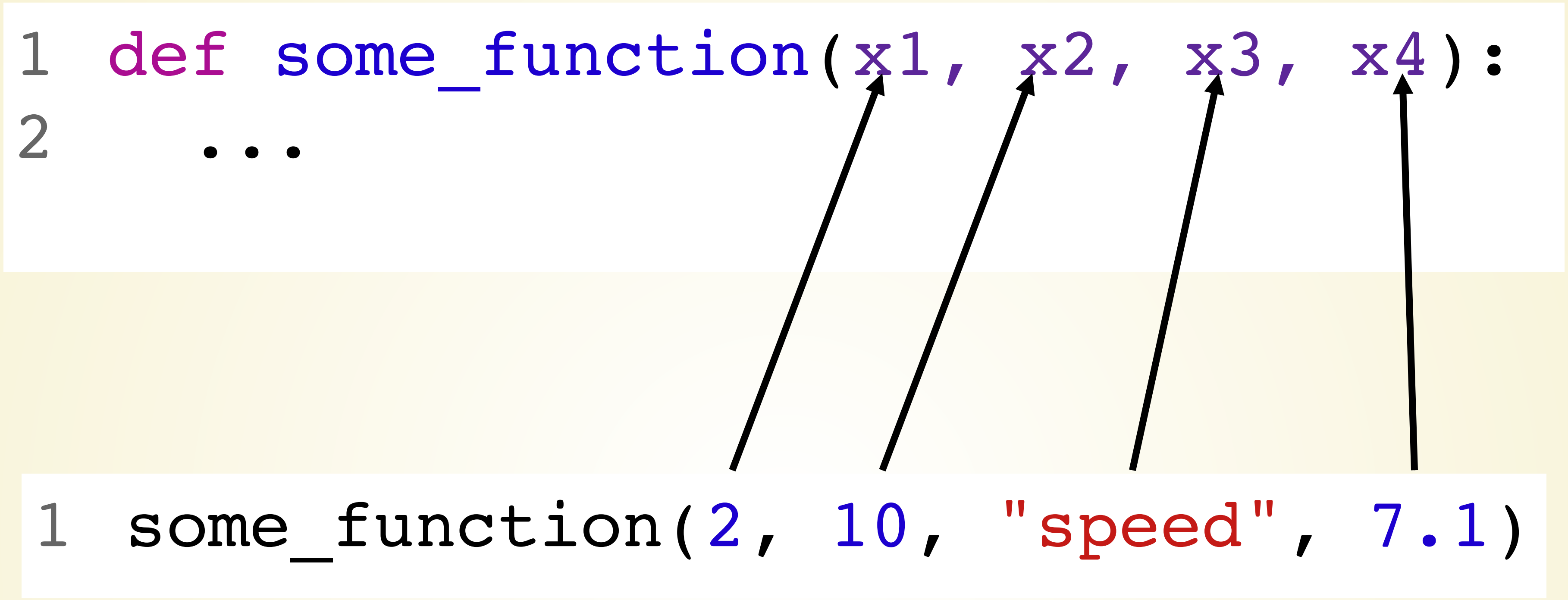
Now we just need to figure out if a number is prime.

Final Example

Let's say we just want to find and print k primes, starting from a given number.

Positional Arguments

and kwargs



```python
def some_function(x1, x2, x3, x4):
    ...
```

```python
some_function(2, 10, "speed", 7.1)
```

Arguments are matched to the parameters by position (i.e. where they show up). This is called using **positional** arguments.

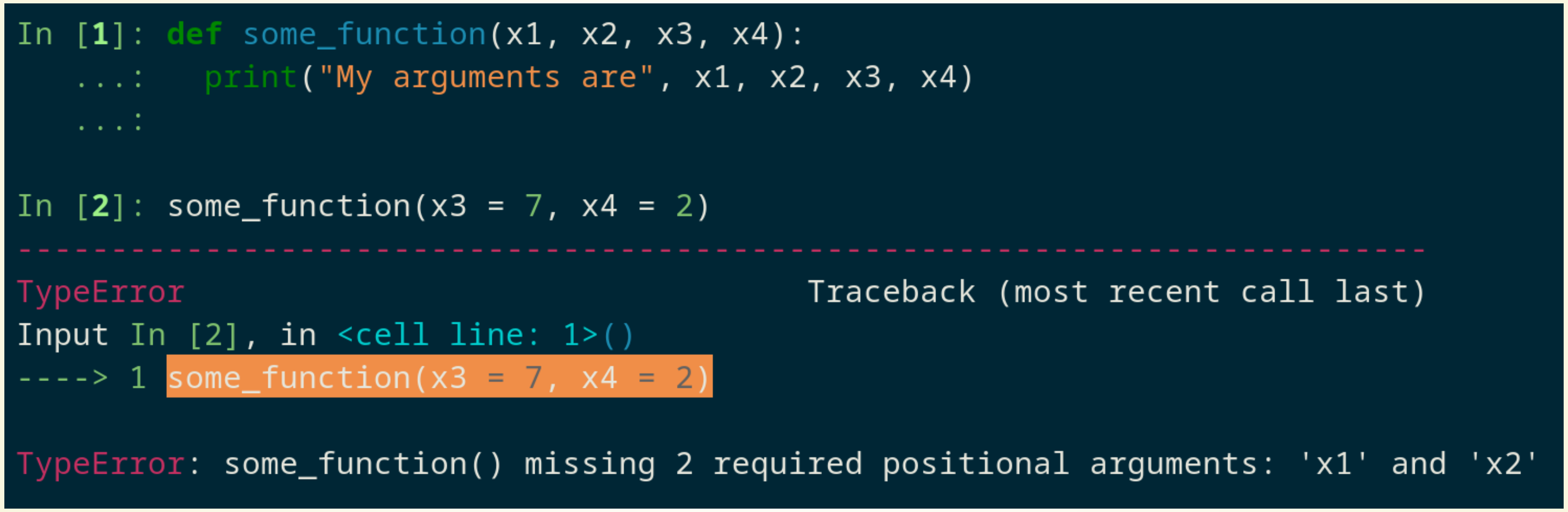
```python
def some_function(x1, x2, x3, x4):
  print("My arguments are", x1, x2, x3, x4)
```

We can also pass parameters by **keyword:**

```python
some_function(5,12,-7, 13)
some_function(x3=-7, x1 = 5, x2 = 12, x4 = 13)
```

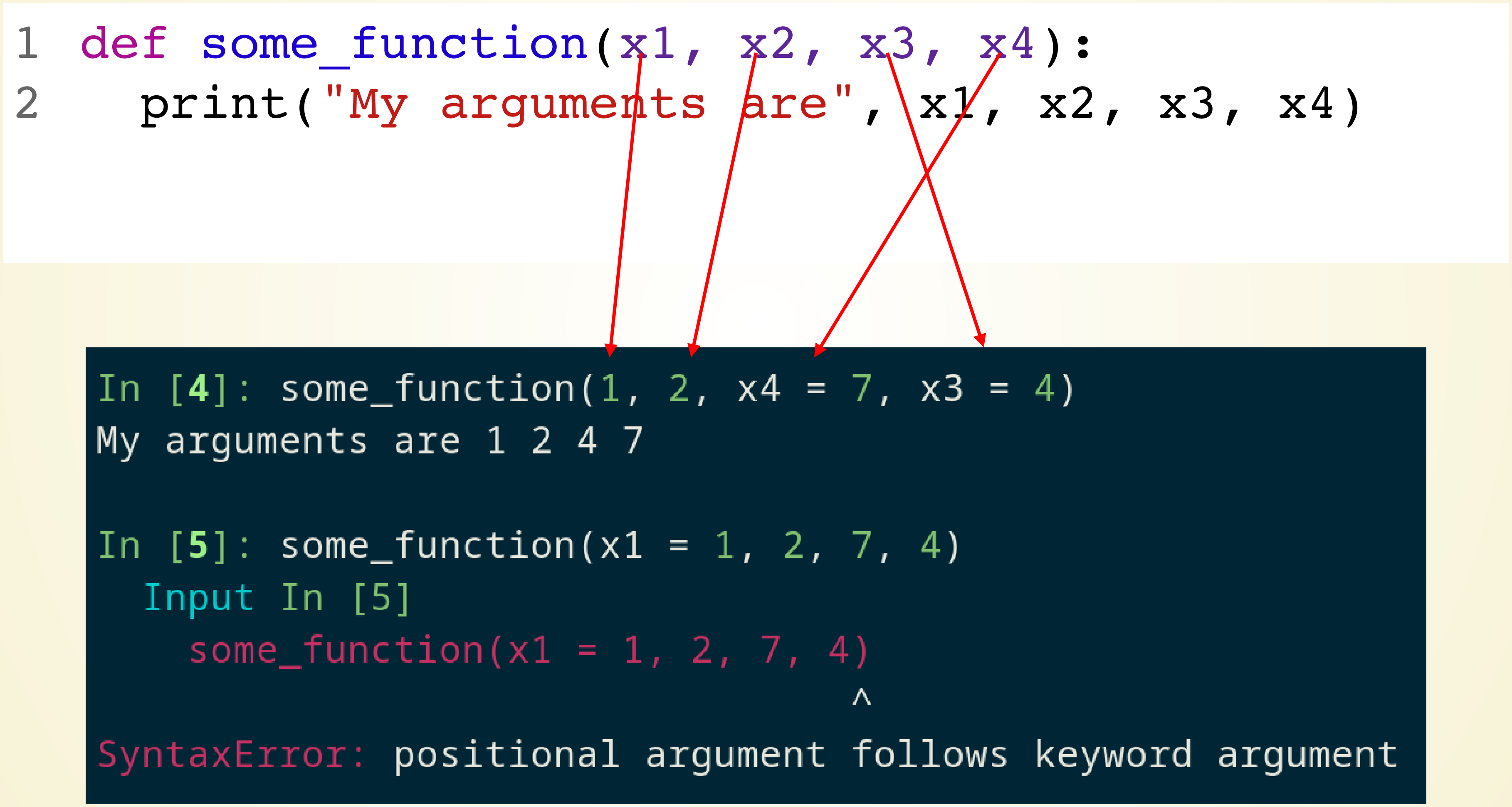
These two calls are equivalent!

You can list the keyword arguments in any order, but they all must still be specified.



```
In [1]: def some_function(x1, x2, x3, x4):
   ...:   print("My arguments are", x1, x2, x3, x4)
   ...:

In [2]: some_function(x3 = 7, x4 = 2)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [2], in <cell line: 1>()
----> 1 some_function(x3 = 7, x4 = 2)

TypeError: some_function() missing 2 required positional arguments: 'x1' and 'x2'
```

You can even mix them, though the positional arguments have to come first.



```python
def some_function(x1, x2, x3, x4):
  print("My arguments are", x1, x2, x3, x4)
```

```
In [4]: some_function(1, 2, x4 = 7, x3 = 4)
My arguments are 1 2 4 7

In [5]: some_function(x1 = 1, 2, 7, 4)
  Input In [5]
    some_function(x1 = 1, 2, 7, 4)
                                ^
SyntaxError: positional argument follows keyword argument
```

Default Arugments

You can also specify **default arguments** for a function. If you don't specify a corresponding argument, the default is used.

```python
def print_rectangle_area(width = 1.0, height = 2.0):
  area = width * height
  print("A rectangle with width", width,
          "and height", height,
         "has area", area)
```

```python
# What do these print?
print_rectangle_area()
print_rectangle_area(4.5, 7.6)
print_rectangle_area(height = 20.5, width = 5.2)
print_rectangle_area(4.5)
print_rectangle_area(height = 10.0)
print_rectangle_area(width = 5.25)
```

You can mix default and non-default arguments in the definition of a function. Again, the non-default arguments have to come first.

```
def email(address, messsage=""):     #Okay!
  ...


def email(message="", address):      #Not okay!
  ...
```

Have any of the built-in functions we've used so far had default arguments?

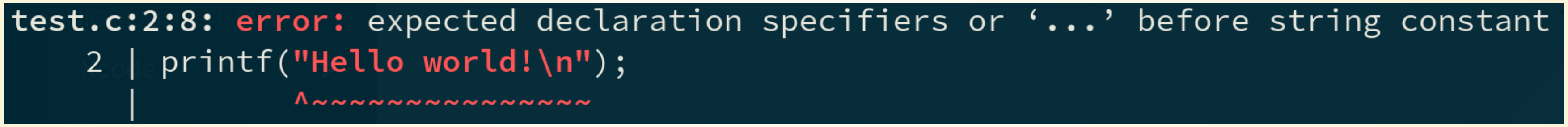
main()

Python lets us write code outside of functions.

This is actually somewhat unusual!

```python
x = 3 + 5
print("Hello world!")
```

```c
#include<stdio.h>
int x = 3 + 5;
printf("Hello world!\n");
```



```
test.c:2:8: error: expected declaration specifiers or '...' before string constant
    2 | printf("Hello world!\n");
      |        ^~~~~~~~~~~~~~~~
```

```python
import math
def main():
    x = int(input("Enter a number: "))

    is_prime = x % 2 != 0
    divisor = 3
    limit = math.sqrt(x)
    while divisor < limit and is_prime:
        if x % divisor == 0:
            is_prime = False
        divisor += 2

    if is_prime:
        print(x, "is prime.")
    else:
        print(x, "is not prime.")

main()
```

Error Handling

```python
inp = int(input("Command: "))

if inp != "quit":
  cmd_number = int(inp)
  if cmd_number != 0:
    cmd = get_command(cmd_number)
    if cmd != "invalid":
      print("You entered", cmd, ". Executing now...")
      exec_cmd(cmd)
    else:
      print("Command lookup gave an invalid command")
  else:
    print("You entered command 0, which cannot be looked up")
else:
  print("Goodbye!")
```

Error Handling

```python
inp = int(input("Command: "))

if inp == "quit":
  print("Goodbye!")
  return

cmd_number = int(inp)
if cmd_number == 0:
  print("You entered command 0, which cannot be looked up")
  return

cmd = get_command(cmd_number)
if cmd == "invalid":
  print("Command lookup gave an invalid command")
  return

print("You entered", cmd, ". Executing now...")
exec_cmd(cmd)
```

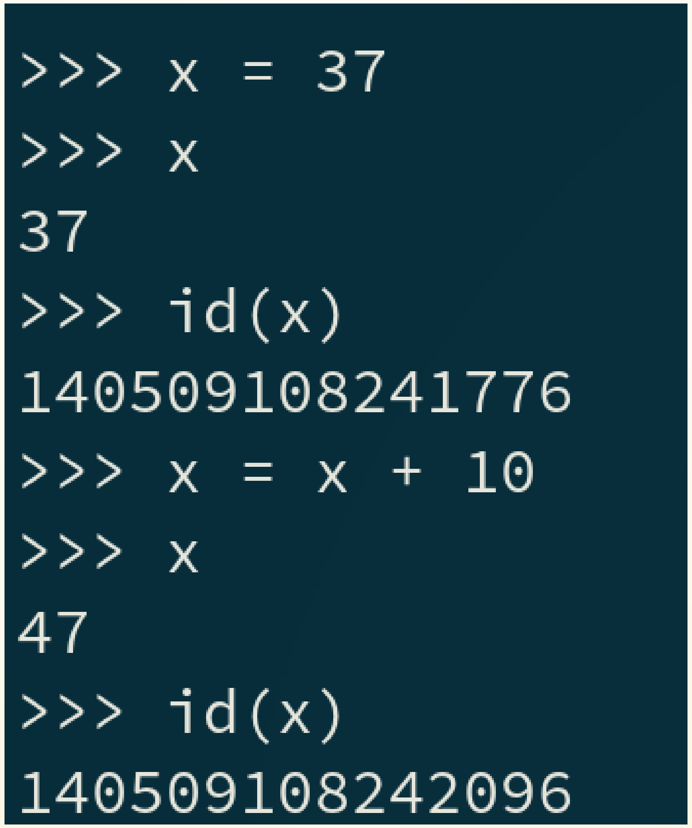
What's our big problem here?

Error Handling

```python
def main():
  inp = int(input("Command: "))

  if inp == "quit":
    print("Goodbye!")
    return

  cmd_number = int(inp)
  if cmd_number == 0:
    print("You entered command 0, which cannot be looked up")
    return

  cmd = get_command(cmd_number)
  if cmd == "invalid":
    print("Command lookup gave an invalid command")
    return

  print("You entered", cmd, ". Executing now...")
  exec_cmd(cmd)

main()
```

There are two kinds of Python objects:

- **mutable**: you can change them in your program
- **immutable:** you cannot change them in your program



```
>>> x = 37
>>> x
37
>>> id(x)
140509108241776
>>> x = x + 10
>>> x
47
>>> id(x)
140509108242096
```

Actually makes a new copy!

Data Type	Description	Example	Mutability
int	Integer. A counting number of unlimited value.	42	immutable
float	A real number.	3.1415926	immutable
str	A sequence of characters. Usually used to model text.	"Hello world"	immutable
bool	A truth value (True or False)	True, False	immutable
tuple	Immutable sequence of mixed types	(4.0, True)	immutable
list	Mutable sequence of mixed types	[1,2,3,4,5]	mutable
set	Collection without duplicates		mutable
dict	A map from keys onto values	{'a': 3, 'b': 4}	mutable

If you pass a mutable object as an argument, it can be changed by your function

If you pass an immutable object, it can't be changed.

```python
def increment(x):
  x += 1
  print(f"Value of x in the function: {x}")

x = 3
print(f"x is {x}")
increment(x)
print(f"x after function call: {x}")
```

```python
def reverse_list(l):
  l.reverse()
  print(f"List in the function: {l}")

l = [1, 2, 3, 4, 5]
print(f"List is {l}")
reverse_list(l)
print(f"List after function call: {l}")
```

Last Bits

Scope

A variable in python has a **scope**, or a region where it's valid.

```python
def main():
    x = 3
    y = 4
    f1(x)

def f1(x):
    return x + y
```

A **global variable** is defined outside of a function and is visible everywhere after it's defined. Using these is usually considered bad practice.

A **local variable** is defined inside of a function, and is visible from its definition until the end of the function.

A local definition **overrides** a global one.

```python
x = 1

def func():
    x = 2
    print(x)


func()
print(x)
```

Returning Multiple Values

```python
def increment_multiple(x, y):
  return (x + 1, y + 1)

x1, x2 = increment_multiple(4, 5.2)
print(f"x1 is {x1} and x2 is {x2}")
```