How to pass a variable number of arguments to a function?

Kirill Smirnov
Data Science Consultant, Altran
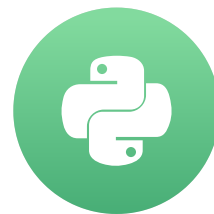
Argument types

There are two types of arguments:

-
-

Argument types

There are two types of arguments:

- positional arguments

-

Argument types

There are two types of arguments:

- positional arguments

- keyword arguments

Argument types

There are two types of arguments:

- **positional arguments**

- keyword arguments

Positional arguments

```python
def func_with_pos_args(arg1, arg2):

    pass
```

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

```python
multiply(2, 3)
```

```
6
```

*args

```python
def func_with_var_pos_args(*args):


    pass
```

```python
func_with_var_pos_args(1, 2, 'hello')
```

DataCamp

*args

```python
def func_with_var_pos_args(*args):


    print(args)
```

```python
func_with_var_pos_args(1, 2, 'hello')
```

```
(1, 2, 'hello')
```

*args

```python
def func_with_var_pos_args(*args):
    for arg in args:
        print(arg)
```

```python
func_with_var_pos_args(1, 2, 'hello')
```

```
1
2
'hello'
```

Redefining multiply()

```python
def multiply(*args):
    result = 1
    for arg in args:
        result = result * arg
    return result
```

```python
multiply(1, 2, 3):
```

```
6
```

```python
multiply(1, 2, 3, 4)
```

```
24
```

Redefining multiply()

```python
def multiply(*nums):
    result = 1
    for num in nums:
        result = result * num
    return result
```

```python
multiply(1, 2, 3):
```

```
6
```

```python
multiply(1, 2, 3, 4)
```

```
24
```

Another use of single asterisk *

```python
def multiply(num1, num2, num3):
    return num1 * num2 * num3
```

```python
multiply(1, 2, 3)
```

```
6
```

Another use of single asterisk *

```python
def multiply(num1, num2, num3):
    return num1 * num2 * num3
```

```python
nums = (2, 3, 4)
```

```python
multiply(*nums)
```

```
24
```

```python
nums = [2, 3]
multiply(*nums, 4)
```

```
24
```

Another use of single asterisk *

```python
def multiply(*args):
    result = 1
    for arg in args:
        result = result * num
    return result
```

```python
nums = (2, 3, 4, 5)
```

```python
multiply(*nums)
```

```
120
```

Argument types

There are two types of arguments:

- positional arguments

- **keyword arguments**

Keyword arguments

```python
def func_with_kwargs(arg1=1, arg2=2):
```

```python
def multiply(x=1, y=2):
    print(str(x) + ' : ' + str(y))
```

```python
multiply(2, 3)
```

```
2 : 3
```

```python
multiply()
```

```
1 : 2
```

Keyword arguments

```python
def func_with_kwargs(arg1=1, arg2=2):
```

```python
def multiply(x=1, y=2):
    print(str(x) + " : " + str(y))
```

```python
multiply(y=5, x=3)
```

```
3 : 5
```

**kwargs

kwargs - keyword arguments

```python
def func_with_var_kwargs(**kwargs):
    print(kwargs)
```

```python
func_with_var_kwargs(arg1=1, arg2=2, arg3=3)
```

```
{arg1: 1, arg2: 2, arg3: 3}
```

```python
func_with_var_kwargs(1, arg2=2, arg3=3)
```

```
TypeError
```

Redefining multiply()

```python
def multiply_kwargs(**kwargs):
    result = 1
    for (key, value) in kwargs.items():
        print(key + ' = ' + str(value))
        result = result * value
    return result
```

```python
def multiply(*args):
    result = 1
    for arg in args:
        result = result * arg
    return result
```

Calling multiply_kwargs()

```
multiply_kwargs(num1=1, num2=2, num3=3, num4=4)
```

```
num1 = 1
num2 = 2
num3 = 3
num4 = 4
24
```

Another use of double asterisk **

```python
def multiply(num1=1, num2=2, num3=3):
    print('num1 = ' + str(num1))
    print('num2 = ' + str(num2))
    print('num3 = ' + str(num3))
    return num1 * num2 * num3
```

```python
multiply()
```

```
num1 = 1
num2 = 2
num3 = 3
6
```

Another use of double asterisk **

```python
def multiply(num1=1, num2=2, num3=3):
    print('num1 = ' + str(num1))
    print('num2 = ' + str(num2))
    print('num3 = ' + str(num3))
    return num1 * num2 * num3
```

```python
nums = {'num1': 10, 'num2': 20, 'num3': 30
```

```python
multiply(**nums)
```

```
num1 = 10
num2 = 20
num3 = 30
6000
```

Another use of double asterisk **

```python
def multiply(num1=1, num2=2, num3=3):
    print('num1 = ' + str(num1))
    print('num2 = ' + str(num2))
    print('num3 = ' + str(num3))
    return num1 * num2 * num3
```

```python
nums = {'num1': 10, 'num3': 30}
```

```python
multiply(**nums)
```

```
num1 = 10
num2 = 2
num3 = 30
600
```

Another use of double asterisk **

```python
def multiply(num1=1, num2=2, num3=3):
    print('num1 = ' + str(num1))
    print('num2 = ' + str(num2))
    print('num3 = ' + str(num3))
    return num1 * num2 * num3
```

```python
nums = {'NUM10': 1, 'num2': 2, 'num3': 3}
```

```python
multiply(**nums)
```

```
TypeError
```

Another use of double asterisk **

```python
def multiply_kwargs(**kwargs):
    result = 1
    for (key, value) in kwargs.items():
        print(key + ' = ' + str(value))
        result = result * value
    return result
```

```python
nums = {
    'num1': 2, 'num2': 3,
    'num3': 4, 'num4': 5
}
```

```python
multiply_kwargs(**nums)
```

```
num1 = 2
num2 = 3
num3 = 4
num4 = 5
120
```

Argument order

```python
def func(                                    ):
```

Argument order

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

- arg1 , arg2 - positional arguments

Argument order

```
def func(arg1, arg2, *args,                    ):
```

- arg1 , arg2 - positional arguments

- *args - positional arguments of variable size

Argument order

```
def func(arg1, arg2, *args, kwarg1, kwarg2,            ):
```

- `arg1` , `arg2` - positional arguments

- `*args` - positional arguments of variable size

- `kwarg1` , `kwarg2` - keyword arguments

Argument order

```python
def func(arg1, arg2, *args, kwarg1, kwarg2, **kwargs):
```

- `arg1` , `arg2` - positional arguments
- `*args` - positional arguments of variable size
- `kwarg1` , `kwarg2` - keyword arguments
- `**kwargs` - keyword arguments of variable size

```python
def func(arg1, arg2, *args):
def func(arg1, arg2, **kwargs):
def func(*args, **kwargs):
```

Let's practice!

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

What is a lambda expression?

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

Kirill Smirnov
Data Science Consultant, Altran

DataCamp

Definition

lambda expression/function - is a short function having the following syntax:

```
lambda arg1, arg2, ...: expression(arg1, arg2, ...)
```

Definition

lambda expression/function - is a short function having the following syntax:

```
lambda
```

Definition

lambda expression/function - is a short function having the following syntax:

```
lambda arg1, arg2, ...:
```

Definition

lambda expression/function - is a short function having the following syntax:

```python
lambda arg1, arg2, ...: expression(arg1, arg2, ...)
```

```python
lambda x: x**2
```

```python
squared = lambda x: x**2

squared(4)
```

```
16
```

4 $\rightarrow$ x $\rightarrow$ x**2 $\rightarrow$ 16

Definition

lambda expression/function - is a short function having the following syntax:

```python
lambda arg1, arg2, ...: expression(arg1, arg2, ...)
```

```python
power = lambda x, y: x**y

power(2, 3)
```

```
8
```

2, 3 → x, y → x**y → 8

Missing argument

```python
power = lambda x, y: x**y
```

```python
power(2)
```

```
TypeError
```

Comparison to normal function definition

```python
squared_lambda = lambda x: x**2
```

```python
def squared_normal(x):
    return x**2
```

DataCamp

Comparison to normal function definition

```
        lambda
```

```
def
```

Comparison to normal function definition

```python
squared_lambda = lambda
```

```python
def squared_normal
```

Comparison to normal function definition

```python
squared_lambda = lambda x:
```

```python
def squared_normal(x):
```

Comparison to normal function definition

```python
squared_lambda = lambda x: x**2
```

```python
def squared_normal(x):
    return x**2
```

```python
squared_lambda(3)
```

```
9
```

```python
squared_normal(3)
```

```
9
```

Passing lambda function as an argument

```python
def function_with_callback(num, callback_function):
    return callback(num)
```

`callback_function(arg)` - a function with one argument

```python
def squared_normal(x):
    return x**2
```

```python
function_with_callback(2, squared_normal)
```

```
4
```

Passing lambda function as an argument

```python
def function_with_callback(num, callback_function):
    return callback(num)
```

`callback_function(arg)`   - a function with one argument

```python
--> def squared_normal(x): <--
-->     return x**2          <--
```

```python
--> function_with_callback(2, squared_normal) <--
```

4

Passing lambda function as an argument

```python
def function_with_callback(num, callback_function):
    return callback(num)
```

`callback_function(arg)` - a function with one argument

```python
function_with_callback(2, lambda x: x**2)
```

```
4
```

Definition

lambda expression/function - is a short function having the following syntax:

```python
lambda arg1, arg2, ...: expression(arg1, arg2, ...)
```

Definition

lambda expression/function - is a short (anonymous) function having the following syntax:

```python
lambda arg1, arg2, ...: expression(arg1, arg2, ...)
```

```python
squared = lambda x: x**2
squared(3)
```

```
9
```

Definition

lambda expression/function - is a short (anonymous) function having the following syntax:

```python
lambda arg1, arg2, ...: expression(arg1, arg2, ...)
```

```python
(lambda x: x**2)(3)
```

```
9
```

Ternary operator

```python
def odd_or_even(num):
    if num % 2 == 0:
        return 'even'
    else:
        return 'odd'
```

```python
odd_or_even(3)
```

```
'odd'
```

```python
odd_or_even(6)
```

```
'even'
```

Ternary operator

```python
def odd_or_even(num):
    return 'even' if num % 2 == 0 else 'odd'
```

```python
odd_or_even(3)
```

```
'odd'
```

```python
odd_or_even(6)
```

```
'even'
```

Ternary operator

```python
odd_or_even = lambda num: 'even' if num % 2 == 0 else 'odd'
```

```python
odd_or_even(3)
```

```
'odd'
```

```python
odd_or_even(6)
```

```
'even'
```

Practical use

Use lambda expressions when it is really necessary!

- within function bodies to perform a small task

- as callbacks

Let's practice!

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

What are the functions map(), filter(), reduce()?

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

Kirill Smirnov
Data Science Consultant, Altran

DataCamp

map()

```
map(                                    )
```

map()

```
map(                           Iterable1, Iterable2, ...)
```

Iterables: `[1, 2, 3, 4, 5]` , `[10, 20, 30, 40, 50]` ,…

DataCamp

map()

```
map(function(x1, x2, ...), Iterable1, Iterable2, ...)
```

Iterables: `[1, 2, 3, 4, 5]` , `[10, 20, 30, 40, 50]` ,…

`1` , `10` ,… → `function(1, 10, ...)` → new object

`2` , `20` ,… → `function(2, 20, ...)` → new object

`3` , `30` ,… → `function(3, 30, ...)` → new object

`4` , `40` ,… → `function(4, 40, ...)` → new object

`5` , `50` ,… → `function(5, 50, ...)` → new object

map() with single Iterable

```
nums = [1, 2, 3, 4, 5]
```

The task is to get `[1, 4, 9, 16, 25]`

```python
def squared(x):
    return x**2
```

```python
squares = map(squared, nums)
```

```python
print(squares)
```

```
<map object at 0x7fdbe4ab3da0>
```

`squares` is iterable

```python
for square in squares:
    print(square)
```

```
1
4
9
16
25
```

map() with single Iterable

```
nums = [1, 2, 3, 4, 5]
```

The task is to get `[1, 4, 9, 16, 25]`

```python
def squared(x):
    return x**2
```

```
squares = map(squared, nums)
```

```
print(squares)
```

```
<map object at 0x7fdbe4ab3da0>
```

`squares` is Iterable

```
list(squares)
```

```
[1, 4, 9, 16, 25]
```

map() with single Iterable

```
nums = [1, 2, 3, 4, 5]
```

The task is to get `[1, 4, 9, 16, 25]`

```python
def squared(x):
    return x**2
```

```
squares = map(squared, nums)
```

```
print(squares)
```

```
<map object at 0x7fdbe4ab3da0>
```

`squares` is Iterator

```
next(squares)
```

```
1
```

```
next(squares)
```

```
4
```

map() with lambda expressions

```
nums = [1, 2, 3, 4, 5]
```

The task is to get `[1, 4, 9, 16, 25]`

```python
def squared(x):
    return x**2
```

```python
squares = map(squared, nums)
list(squares)
```

```
[1, 4, 9, 16, 25]
```

```
nums = [1, 2, 3, 4, 5]
```

The task is to get `[1, 4, 9, 16, 25]`

```python
squares = map(lambda x: x**2, nums)
list(squares)
```

```
[1, 4, 9, 16, 25]
```

map() with multiple Iterables

```python
nums1 = [1, 2, 3, 4, 5]
nums2 = [10, 20, 30, 40, 50]
```

The task is to get: `[1*10, 2*20, 3*30, 4*40, 5*50] = [10, 40, 90, 160, 250]`

```python
mult = map(lambda x, y: x*y, nums1, nums2)
```

```python
list(mult)
```

```
[10, 40, 90, 160, 250]
```

filter()

```
filter(                         )
```

filter()

```
filter(              Iterable)
```

Iterable: `[1, 2, 3, 4, 5]`

filter()

```
filter(function(x), Iterable)
```

Iterable: `[1, 2, 3, 4, 5]`

1 → `function(1)` → `True` → 1 is kept

2 → `function(2)` → `False` → 2 is rejected

3 → `function(3)` → `True` → 3 is kept

4 → `function(4)` → `False` → 4 is rejected

5 → `function(5)` → `True` → 5 is kept

filter() example

```python
nums = [-3, -2, -1, 0, 1, 2, 3]
```

The task is to get: `[1, 2, 3]`

```python
def positive(x):
    return x > 0
```

```python
fobj = filter(positive, nums)
```

```python
print(fobj)
```

```
<filter object at 0x7f196d378d68>
```

`fobj` is Iterable

```python
for item in fobj:
    print(item)
```

```
1
2
3
```

DataCamp

filter() example

```python
nums = [-3, -2, -1, 0, 1, 2, 3]
```

The task is to get: `[1, 2, 3]`

```python
def positive(x):
    return x > 0
```

```python
fobj = filter(positive, nums)
```

```python
print(fobj)
```

```
<filter object at 0x7f196d378d68>
```

`fobj` is Iterable

```python
list(fobj)
```

```
[1, 2, 3]
```

filter() example

```python
nums = [-3, -2, -1, 0, 1, 2, 3]
```

The task is to get: `[1, 2, 3]`

```python
def positive(x):
    return x > 0
```

```python
fobj = filter(positive, nums)
```

```python
print(fobj)
```

```
<filter object at 0x7f196d378d68>
```

`fobj` is Iterator

```python
next(fobj)
```

```
1
```

```python
next(fobj)
```

```
4
```

DataCamp

filter() with lambda expressions

```python
nums = [-3, -2, -1, 0, 1, 2, 3]
```

The task is to get: `[1, 2, 3]`

```python
def positive(x):
    return x > 0
```

```python
fobj = filter(positive, nums)
list(fobj)
```

```
[1, 2, 3]
```

```python
nums = [-3, -2, -1, 0, 1, 2, 3]
```

The task is to get: `[1, 2, 3]`

```python
fobj = filter(lambda x: x > 0, nums)
list(fobj)
```
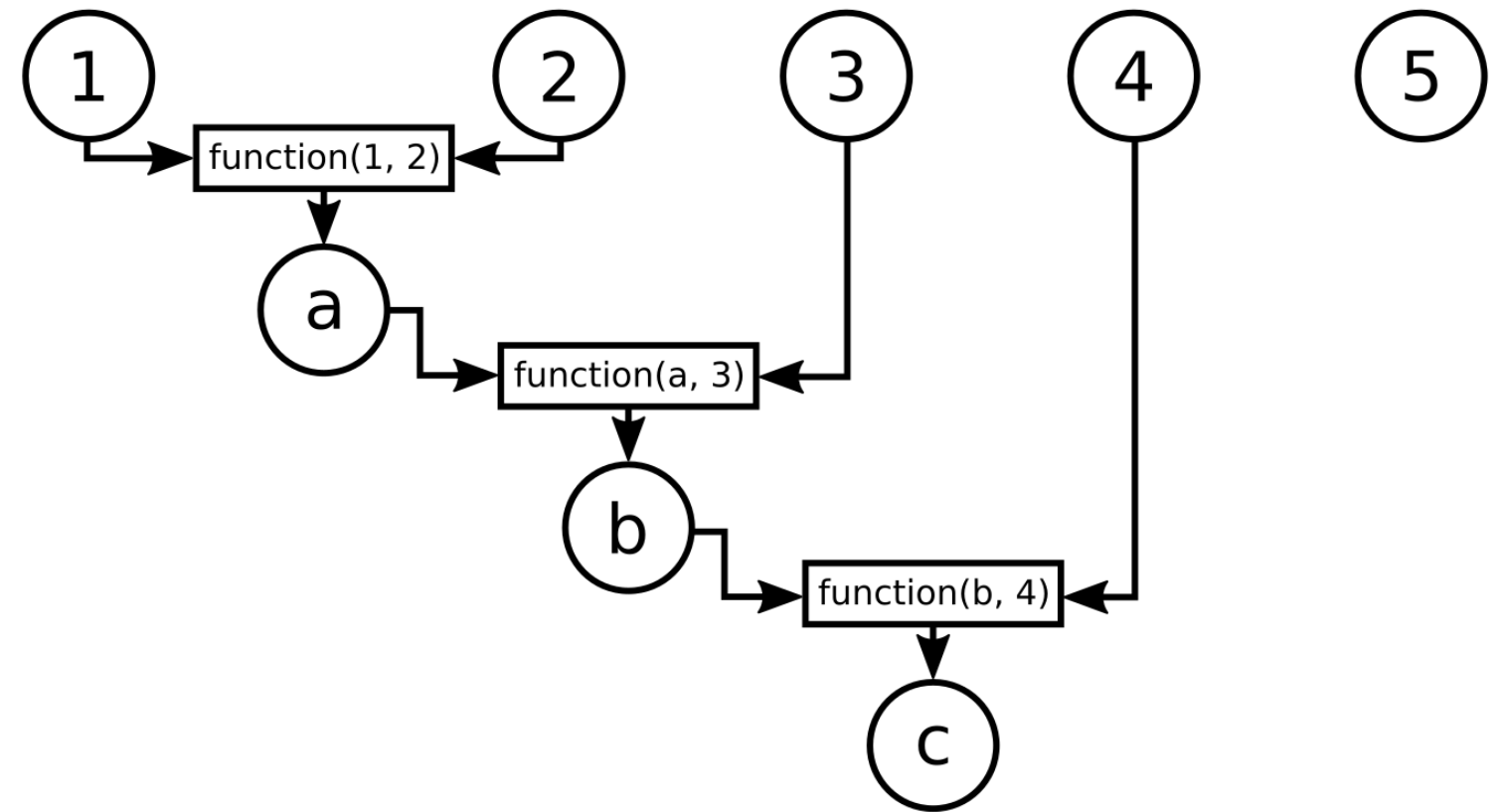
```
[1, 2, 3]
```

reduce()

```
from functools import reduce

reduce(function(x, y), Iterable)
```

Iterable: `[1, 2, 3, 4, 5]`

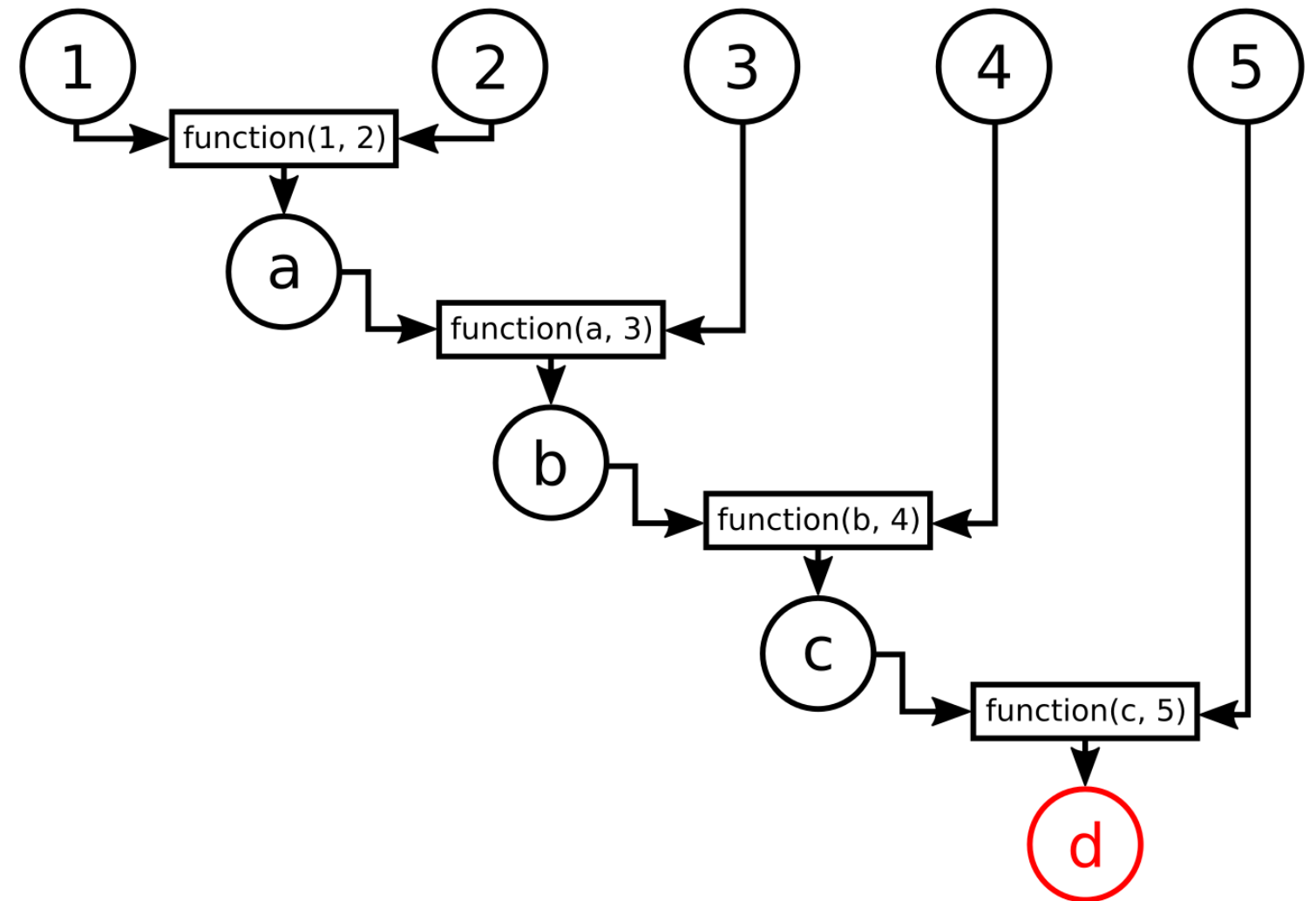`[1, 2, 3, 4, 5]` → new object of the same type as the content

①   ② ③ ④ ⑤

reduce()

```
from functools import reduce

reduce(function(x, y), Iterable)
```

Iterable: `[1, 2, 3, 4, 5]`

`[1, 2, 3, 4, 5]` → new object of the same
type as the content

reduce()

```python
from functools import reduce

reduce(function(x, y), Iterable)
```

Iterable: `[1, 2, 3, 4, 5]`

`[1, 2, 3, 4, 5]` → new object of the same type as the content

reduce()

```
from functools import reduce

reduce(function(x, y), Iterable)
```

Iterable: `[1, 2, 3, 4, 5]`

`[1, 2, 3, 4, 5]` → new object of the same type as the content

reduce()

```
from functools import reduce

reduce(function(x, y), Iterable)
```
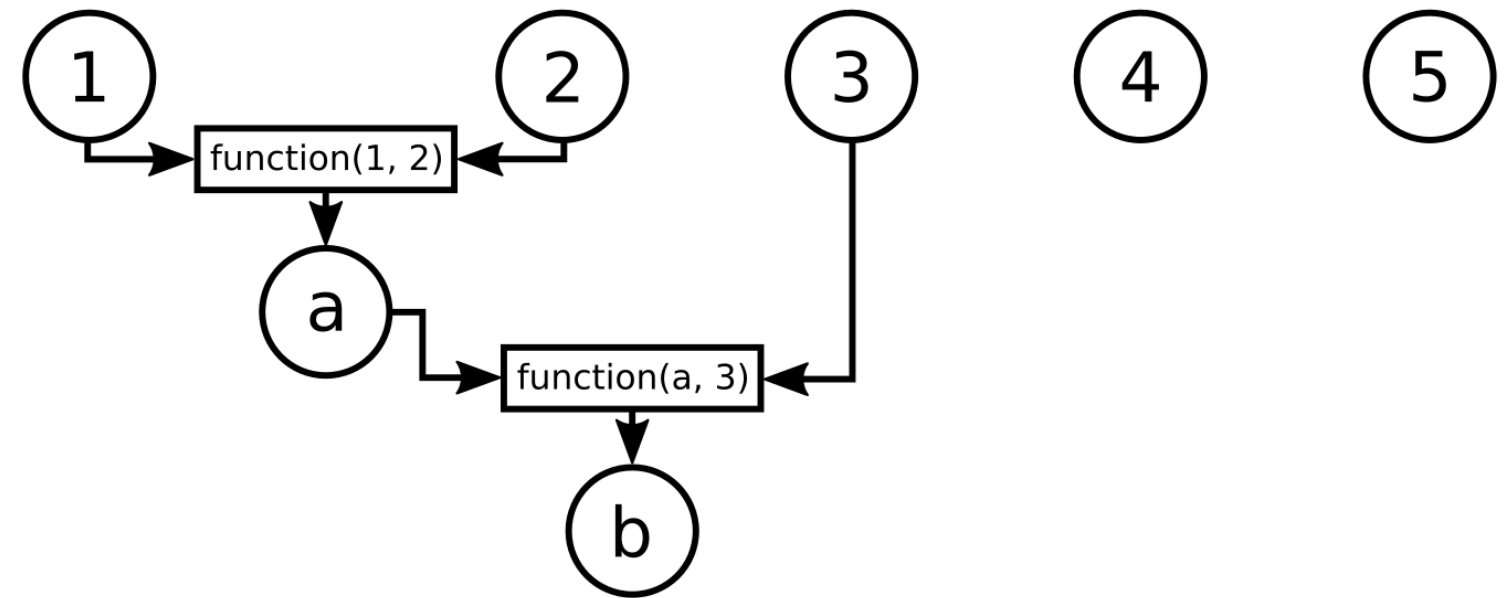
Iterable: `[1, 2, 3, 4, 5]`

`[1, 2, 3, 4, 5]` → new object of the same type as the content

reduce() example

```python
nums = [8, 4, 5, 1, 9]
```

The task is to get: `1` - minimum

```python
def smallest(x, y):
    if x < y:
        return x
    else:
        return y
```

```python
reduce(smallest, nums)
```

```
1
```

smallest(8, 4) $\rightarrow$ 4

smallest(4, 5) $\rightarrow$ 4

smallest(4, 1) $\rightarrow$ 1

smallest(1, 9) $\rightarrow$ 1 - final result

reduce() with lambda expressions

```
nums = [8, 1, 4, 2, 9]
```

The task is to get: 1 - minimum

```python
def smallest(x, y):
    if x < y:
        return x
    else:
        return y
```

```python
reduce(smallest, nums)
```

```
1
```

```
nums = [8, 1, 4, 2, 9]
```

The task is to get: 1 - minimum
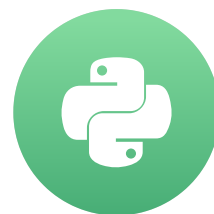
```python
reduce(lambda x: x if x < y else y, nums)
```

```
1
```

Let's practice!

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

Definition

- Recursion is the process of defining a problem in terms of itself

- Recursion is a process in which a function calls itself as a subroutine

Example: Factorial $n!$

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

$n = 4$:

$4! = 4 \cdot 3 \cdot 2 \cdot 1$

$4! = 24$

Factorial - Iterative Approach

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 =$$

$$= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

Iterative solution:

```python
# iterative factorial
def fact_iter(n):
    result = 1
    # looping over numbers from 1 to n
    for num in range(1, n+1)
        result = num * result

    return result
```

$n = 4 :$

result $= 1$

1. result $= 1 *$ result $(1) = 1$
2. result $= 2 *$ result $(1) = 2$
3. result $= 3 *$ result $(2) = 6$
4. result $= 4 *$ result $(4) = 24$

$$4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$

Factorial - Recursive Approach

$$n! = n \cdot (n-1)!$$

```python
def fact_rec(n):
    return n * fact_rec(n-1)
```

What's wrong with that code?

```python
fact_rec(4)
```

```
RecursionError
```

We must define a base case!

$$n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1$$

A stopping criterion / base case: $1! = 1$

```python
def fact_rec(n):
    if n == 1:
        return 1
    return n * fact_rec(n-1)
```
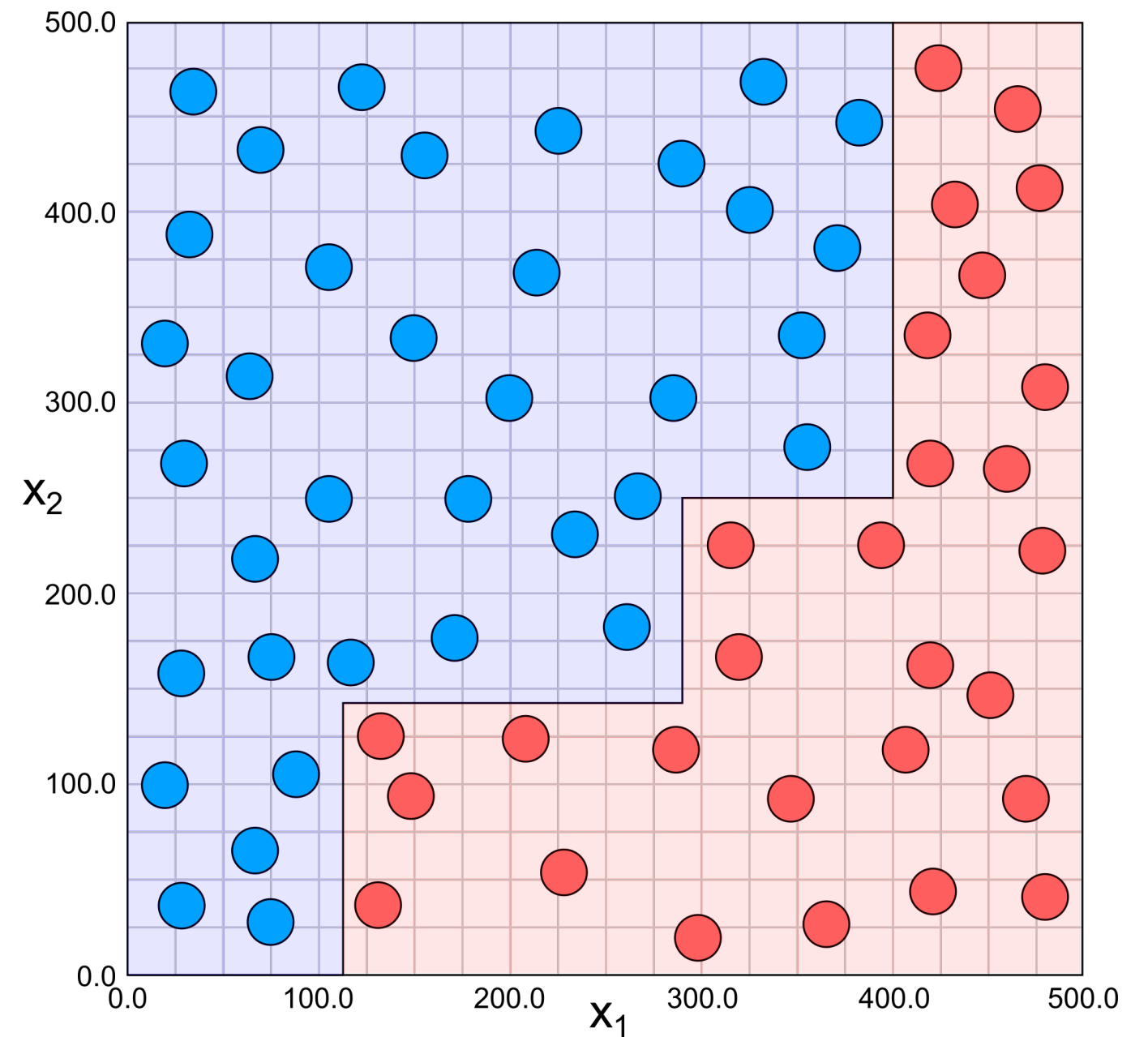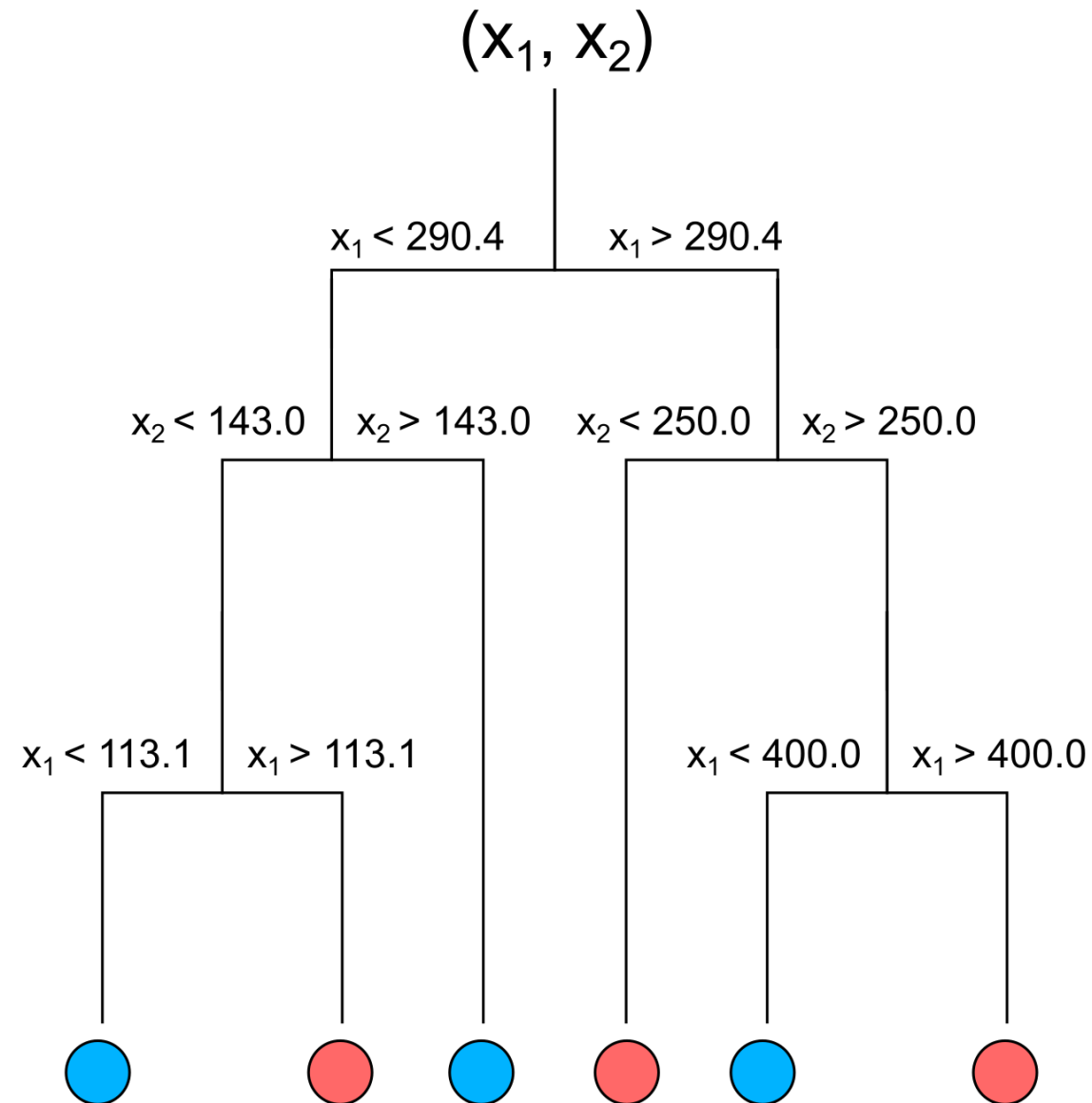
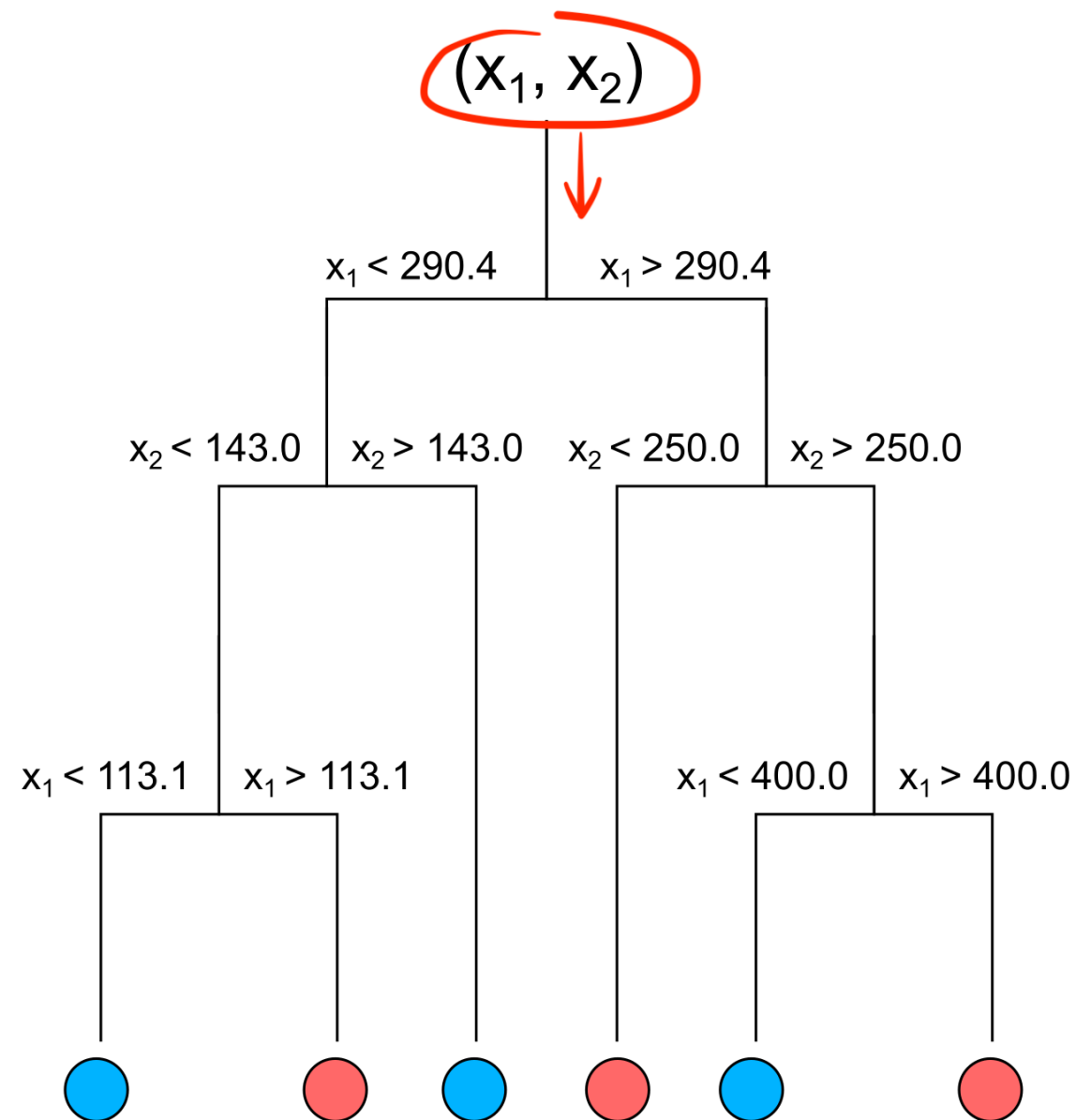```python
fact_rec(4)
```

```
24
```

Wrapping Up

Recursive functions have two main components:

- a recursive call to a smaller problem of itself

- a base case that prevents an infinite calling
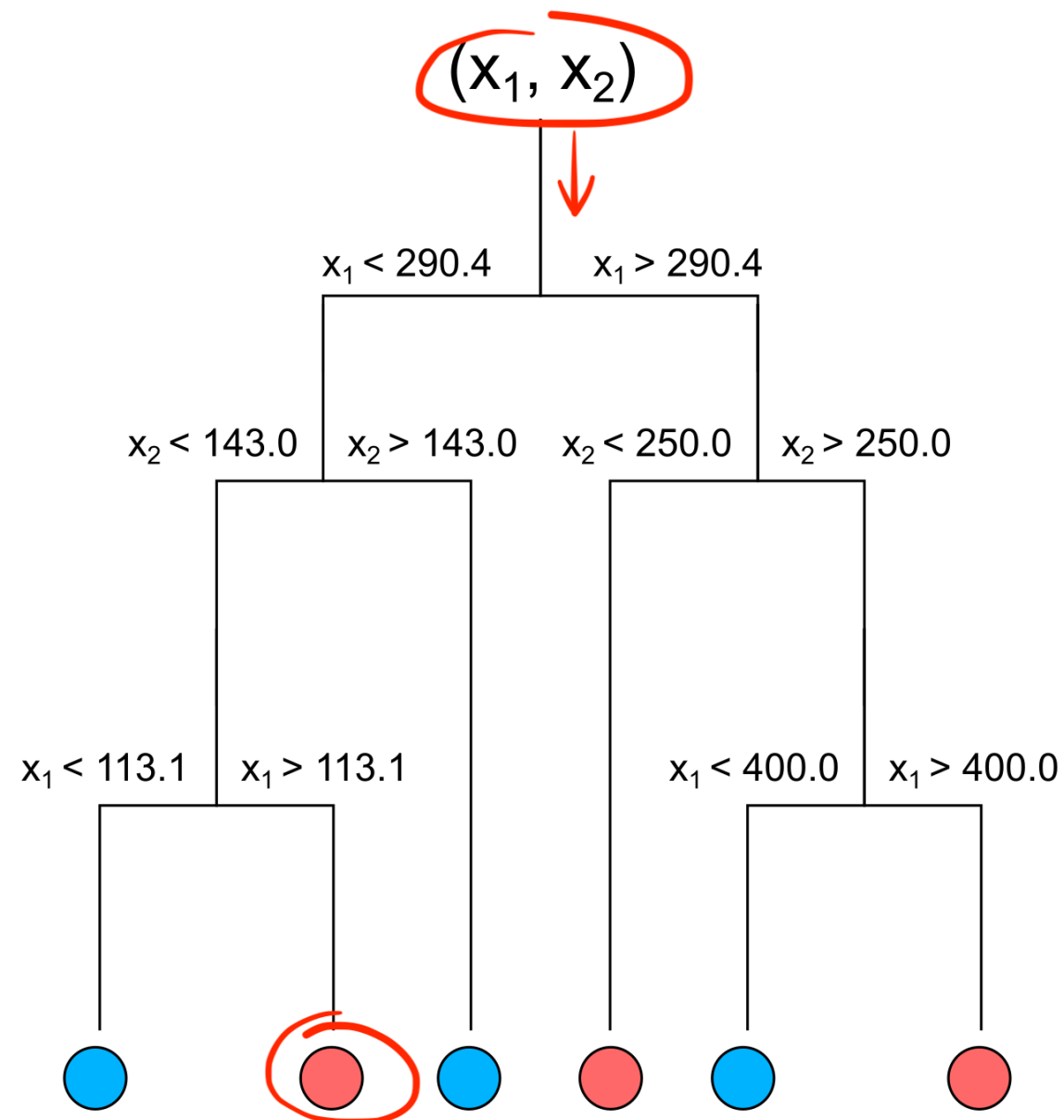
Example - Decision Trees

Traversing a Decision Tree

(x_1, x_2)

$x_1 < 290.4$ $x_1 > 290.4$

$x_2 < 143.0$ $x_2 > 143.0$ $x_2 < 250.0$ $x_2 > 250.0$

$x_1 < 113.1$ $x_1 > 113.1$ $x_1 < 400.0$ $x_1 > 400.0$

x - a new sample (x_1, x_2)

```python
# Pseudo algorithm for finding out the category:

category = pred(node, x):

    # Check if there is a split
    if node.hasSplitting:

        # Check which child node to take
        if node.goToLeftChild(x):
            return pred(node.leftChild, x)
        if node.goToRightChild(x):
            return pred(node.rightChild, x)
```

Traversing a Decision Tree

(x_1, x_2)

$x_1 < 290.4$ $x_1 > 290.4$

$x_2 < 143.0$ $x_2 > 143.0$ $x_2 < 250.0$ $x_2 > 250.0$

$x_1 < 113.1$ $x_1 > 113.1$ $x_1 < 400.0$ $x_1 > 400.0$

x - a new sample (x_1, x_2)

```python
# Pseudo algorithm for finding out the category:

category = pred(node, x):

    # Check if there is a split
    if node.hasSplitting:

        # Check which child node to take
        if node.goToLeftChild(x):
            return pred(node.leftChild, x)
        if node.goToRightChild(x):
            return pred(node.rightChild, x)

    # Returning the category
    return node.category
```

DataCamp

Let's practice!

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON