Lecture 08: Functions in Python

UNIVERSITY of GREENWICH

Alliance with FPT Education

Pearson BTEC

TABLE OF CONTENTS

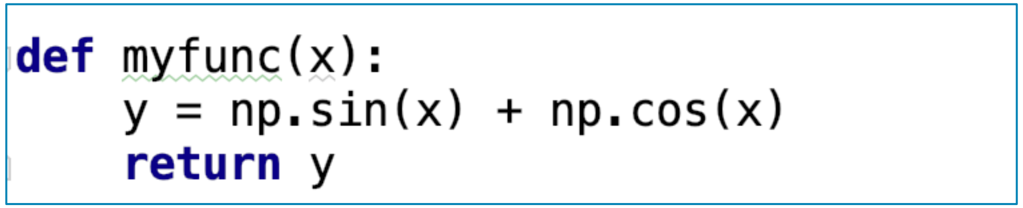
- Define a function
- Keyword arguments
- Return multiple values
- Iterator vs Generator

UNIVERSITY of GREENWICH
Alliance with FPT Education

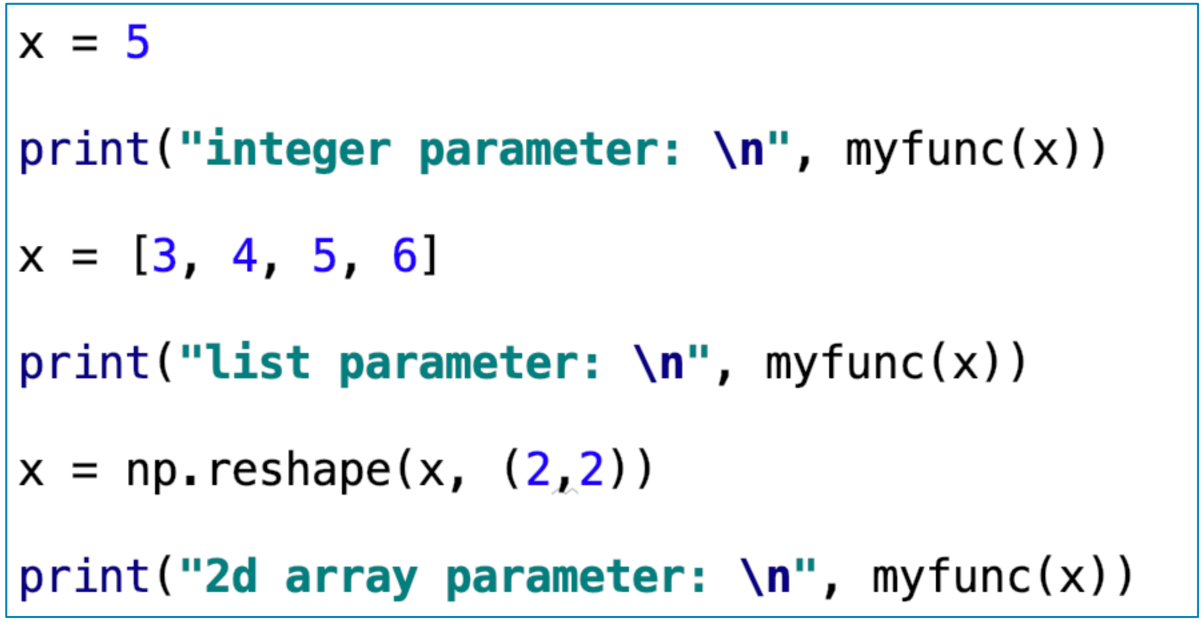
Pearson BTEC

User-defined functions

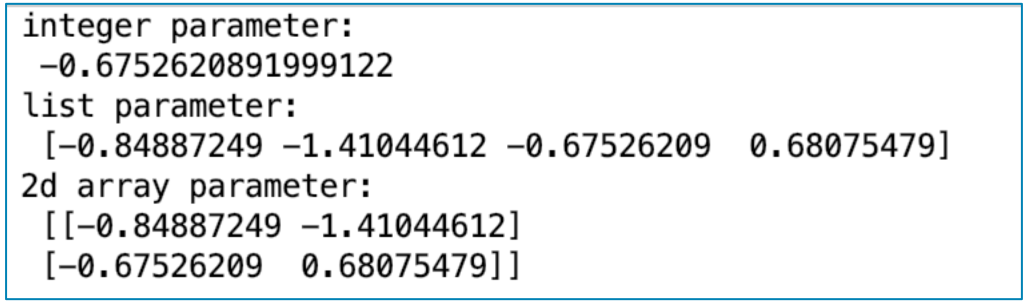
- Very simple with keyword def
- Generic by default



```
def myfunc(x):
    y = np.sin(x) + np.cos(x)
    return y
```

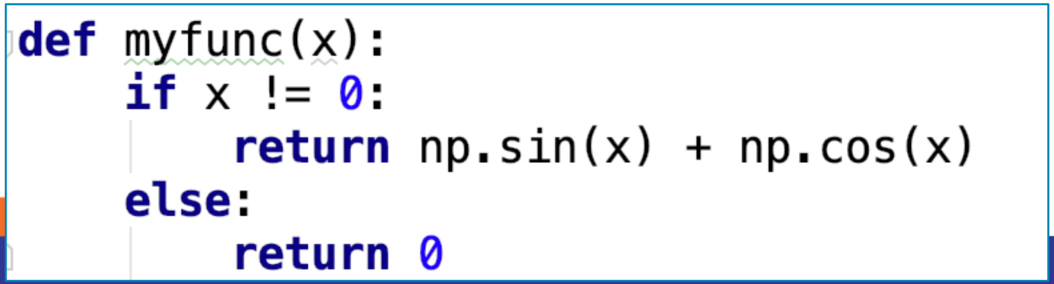


```
x = 5

print("integer parameter: \n", myfunc(x))

x = [3, 4, 5, 6]

print("list parameter: \n", myfunc(x))

x = np.reshape(x, (2,2))

print("2d array parameter: \n", myfunc(x))
```



```
integer parameter:
 -0.6752620891999122
list parameter:
 [-0.84887249 -1.41044612 -0.67526209  0.68075479]
2d array parameter:
 [[-0.84887249 -1.41044612]
 [-0.67526209  0.68075479]]
```

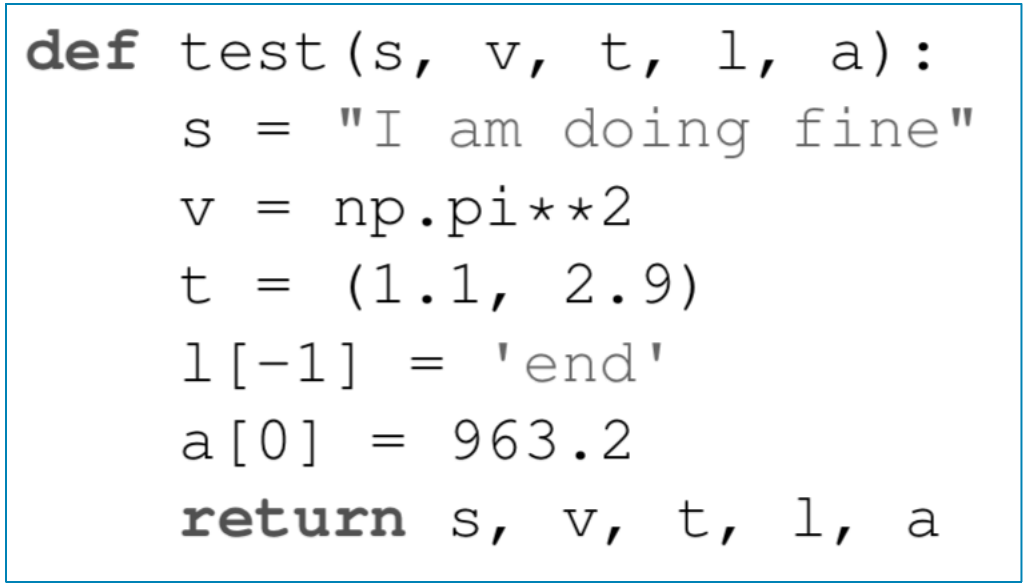
- But not always



```
def myfunc(x):
    if x != 0:
        return np.sin(x) + np.cos(x)
    else:
        return 0
```

Mutable and immutable arguments

- Simple numeric, strings and tuples are immutable
- Lists and arrays are mutable (contents)

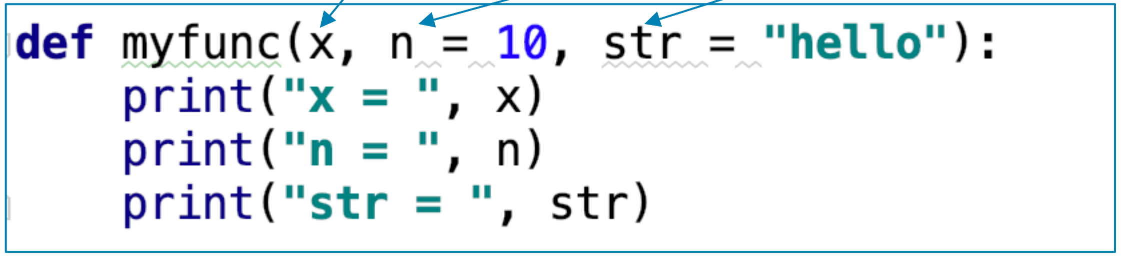


```
def test(s, v, t, l, a):
    s = "I am doing fine"
    v = np.pi**2
    t = (1.1, 2.9)
    l[-1] = 'end'
    a[0] = 963.2
    return s, v, t, l, a
```

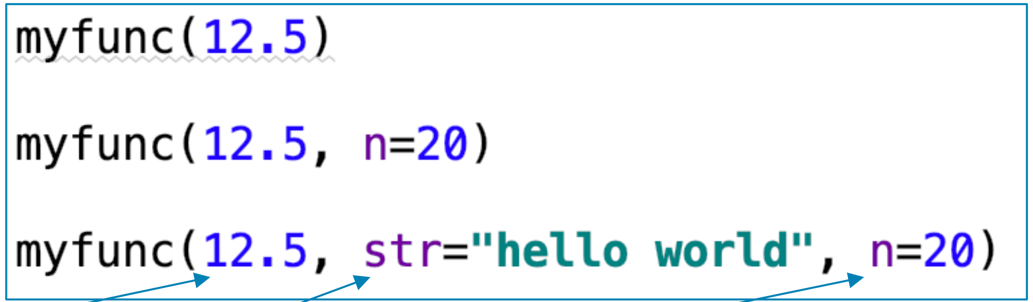
1. Create a string, a float, a tuple, a list and an array
2. Print these values
3. Call test and print returned values
4. Print original values again
5. Which values have changed and why?

Keyword arguments

- Positional arguments vs keyword arguments



```python
def myfunc(x, n = 10, str = "hello"):
    print("x = ", x)
    print("n = ", n)
    print("str = ", str)
```

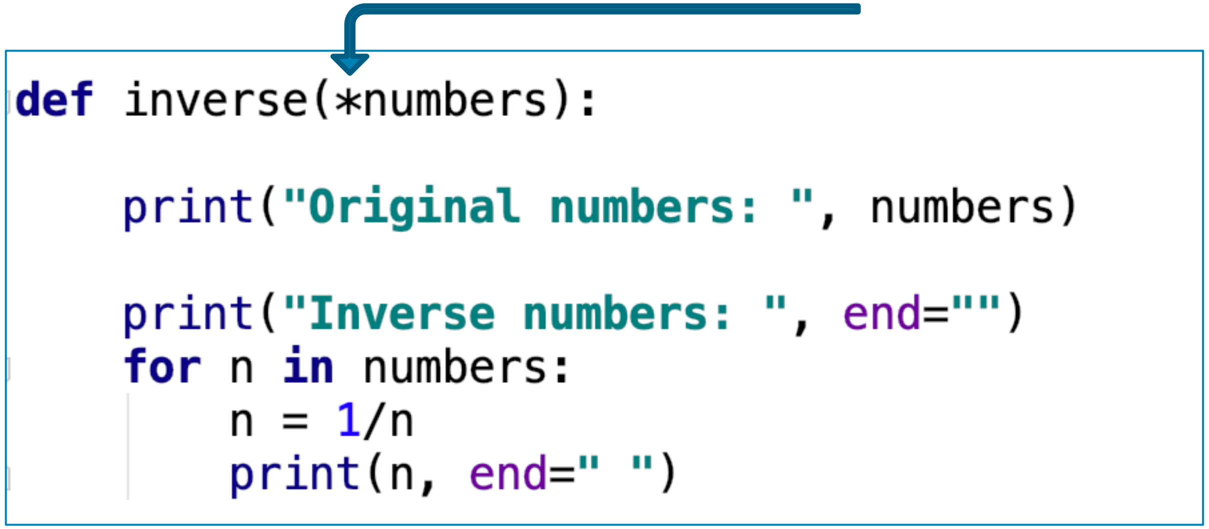


```python
myfunc(12.5)

myfunc(12.5, n=20)

myfunc(12.5, str="hello world", n=20)
```

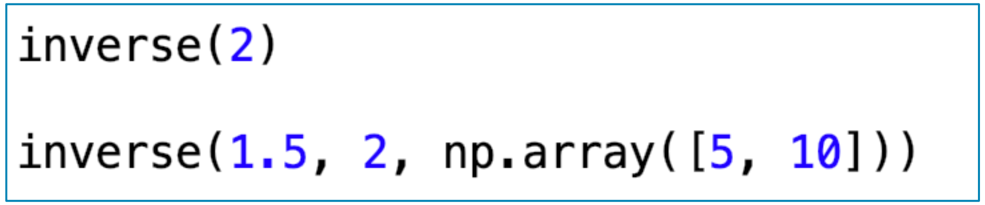
- When using, positional parameters have to appear before any keywords while keywords can be appear in any order

Variable number of arguments

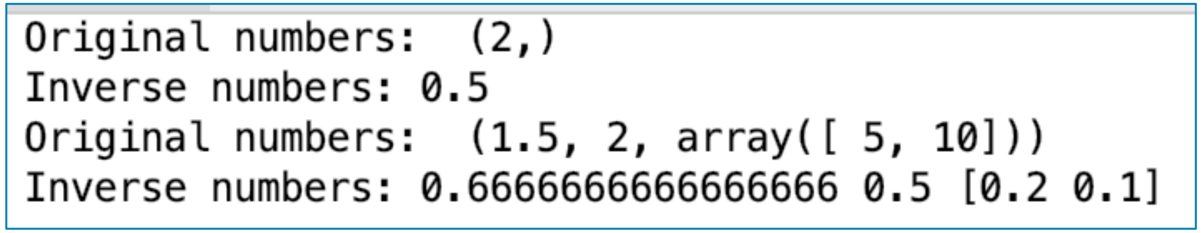
- Function that can pass as many arguments as needed



```python
def inverse(*numbers):

    print("Original numbers: ", numbers)

    print("Inverse numbers: ", end="")
    for n in numbers:
        n = 1/n
        print(n, end=" ")
```



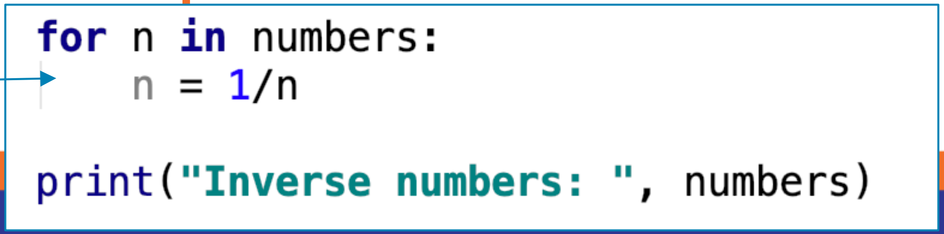
```python
inverse(2)

inverse(1.5, 2, np.array([5, 10]))
```



```
Original numbers:  (2,)
Inverse numbers: 0.5
Original numbers:  (1.5, 2, array([ 5, 10]))
Inverse numbers: 0.6666666666666666 0.5 [0.2 0.1]
```

- Arguments are passed in a tuple

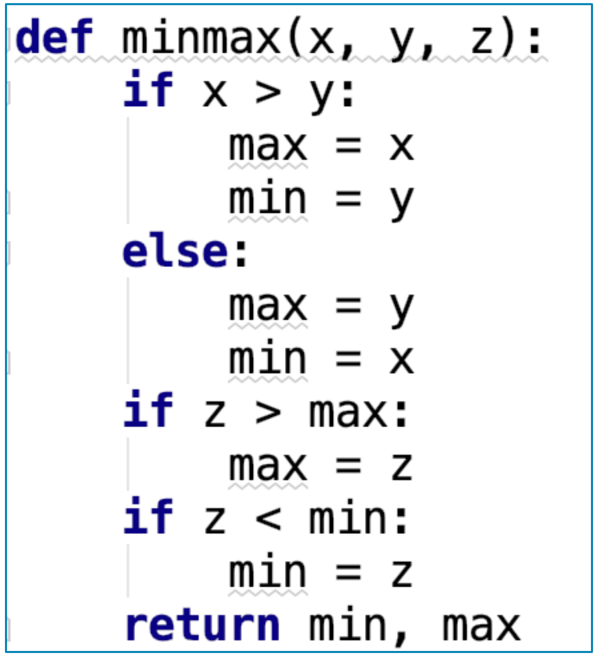
Won’t work as expected!
Why?



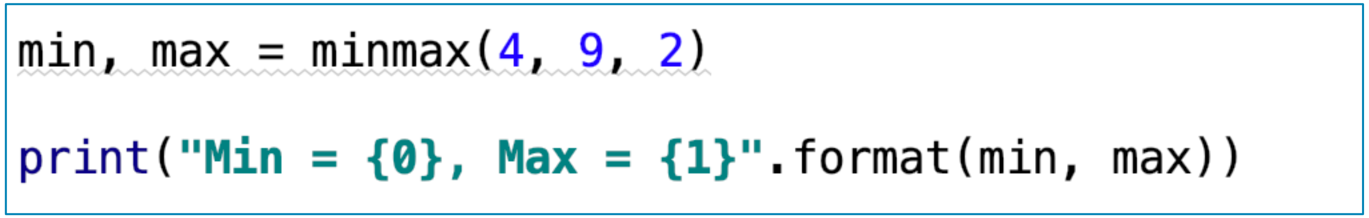
```python
for n in numbers:
    n = 1/n

print("Inverse numbers: ", numbers)
```

Return more than one outputs

- Functions can return more than one outputs
  - – No need for “pass by reference”



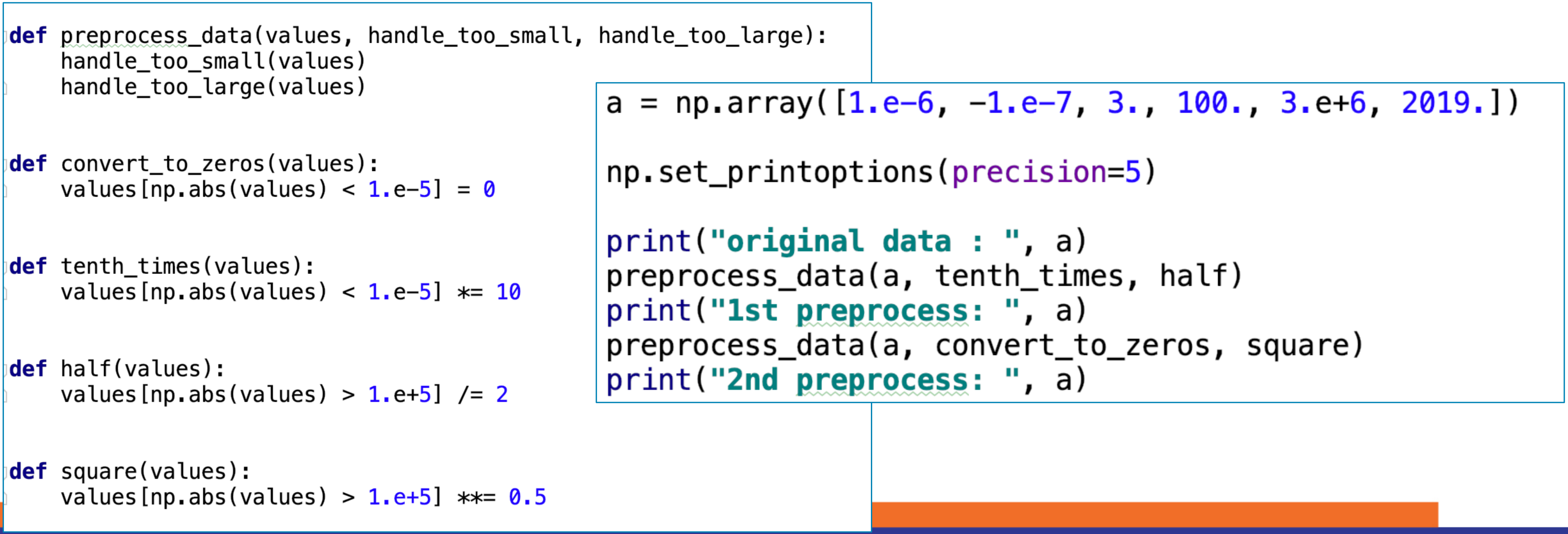
```
def minmax(x, y, z):
    if x > y:
        max = x
        min = y
    else:
        max = y
        min = x
    if z > max:
        max = z
    if z < min:
        min = z
    return min, max
```



```
min, max = minmax(4, 9, 2)

print("Min = {0}, Max = {1}".format(min, max))
```

Function name as argument

- Function can be passed as parameter

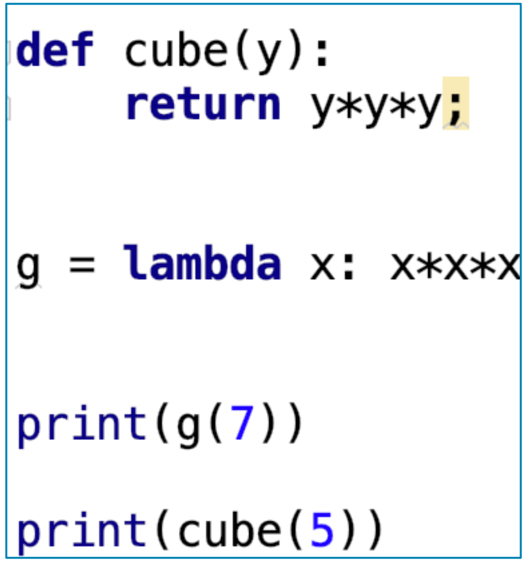


```python
def preprocess_data(values, handle_too_small, handle_too_large):
    handle_too_small(values)
    handle_too_large(values)


def convert_to_zeros(values):
    values[np.abs(values) < 1.e-5] = 0


def tenth_times(values):
    values[np.abs(values) < 1.e-5] *= 10


def half(values):
    values[np.abs(values) > 1.e+5] /= 2


def square(values):
    values[np.abs(values) > 1.e+5] **= 0.5
```

```python
a = np.array([1.e-6, -1.e-7, 3., 100., 3.e+6, 2019.])

np.set_printoptions(precision=5)

print("original data : ", a)
preprocess_data(a, tenth_times, half)
print("1st preprocess: ", a)
preprocess_data(a, convert_to_zeros, square)
print("2nd preprocess: ", a)
```

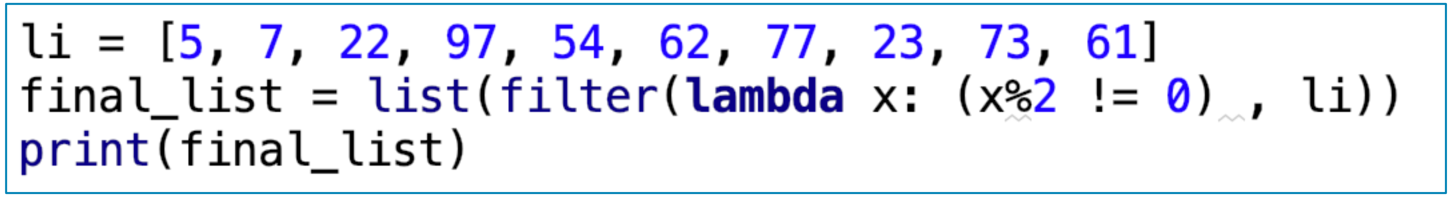
Lambda functions

- Lambda functions are used when in need of a short, no reuse function

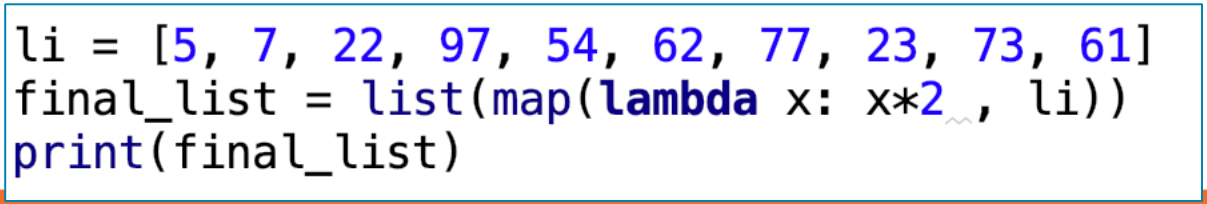


```
def cube(y):
    return y*y*y;


g = lambda x: x*x*x


print(g(7))

print(cube(5))
```

- List doesn’t support 1-by-1 operations (but array does!), so lambda functions can be useful in some cases



```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(filter(lambda x: (x%2 != 0) , li))
print(final_list)
```



```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(map(lambda x: x*2 , li))
print(final_list)
```

exercises

1. Suppose we have a data of 2d list, each row contains name, phone and salary. Write a function to cleansing data by correct name, phone and salary with following options. Each option should be a function.
   a) Empty name can be changed to a default name (John Doe)
   b) Too long name should be truncated (max 20 characters)
   c) Phone must be digits. Any non-digit should change to 0. (i.e 012ab34 => 0120034)
   d) Phone must be digits, any non-digit should change to a previous digit (or 0 if not)
   e) Valid salary is in range [200, 2000]. Any invalid salary should change to 200 or 2000 correspondingly
   f) Valid salary is in range [200, 2000]. Any invalid salary should change to mean of valid salary

exercises

2. Having a list of random integers (100 elements)

a) Find negative numbers and make them positive with the same absolute value

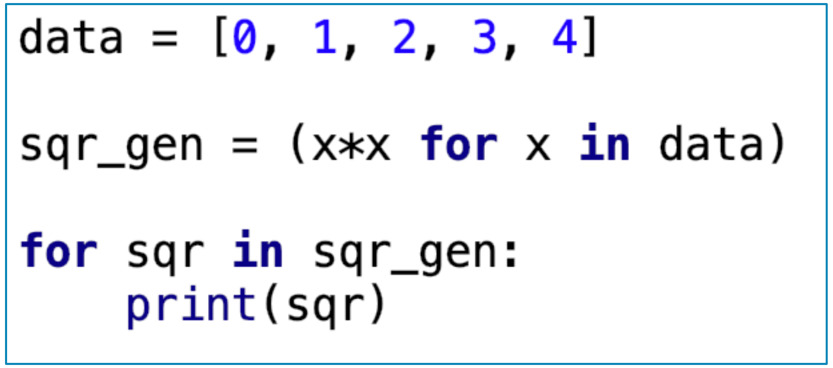
b) Extract prime numbers from the list above. Print result (you may need to write a prime function)

c) For each prime number, print the numbers in the list which is divisible by that prime

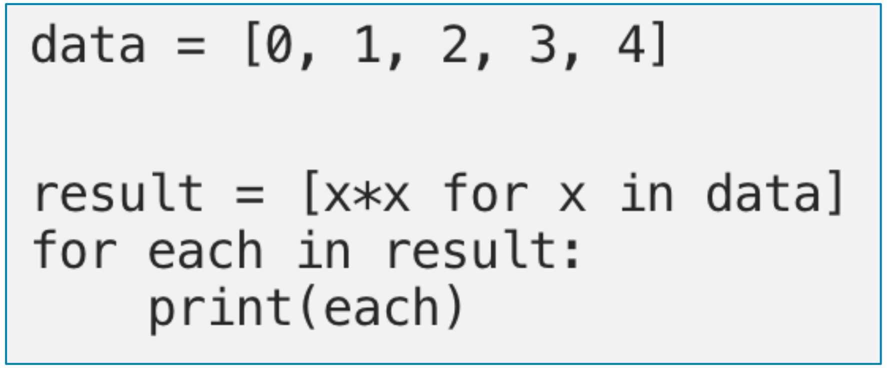
generators

- Generators are functions that allow you to declare a function and have it behave like an iterator. Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

Using generator



```
data = [0, 1, 2, 3, 4]

sqr_gen = (x*x for x in data)

for sqr in sqr_gen:
    print(sqr)
```



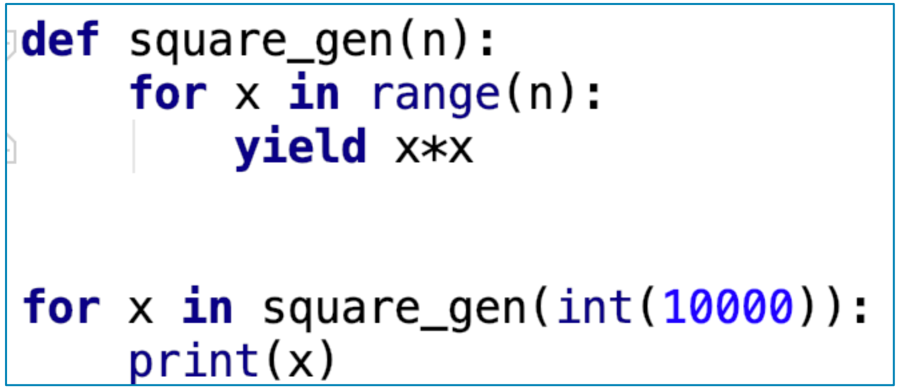
```
data = [0, 1, 2, 3, 4]


result = [x*x for x in data]
for each in result:
    print(each)
```

Using list comprehension

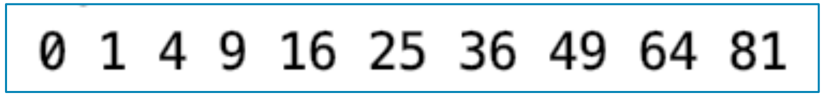
- Unlike a list, a generator only can be used once. When it is empty. It is empty.
- What happens if repeat the for loop again in these 2 codes example above?

generators

- Yield: it’s similar to return keyword but it’s used in generator to return one at a time



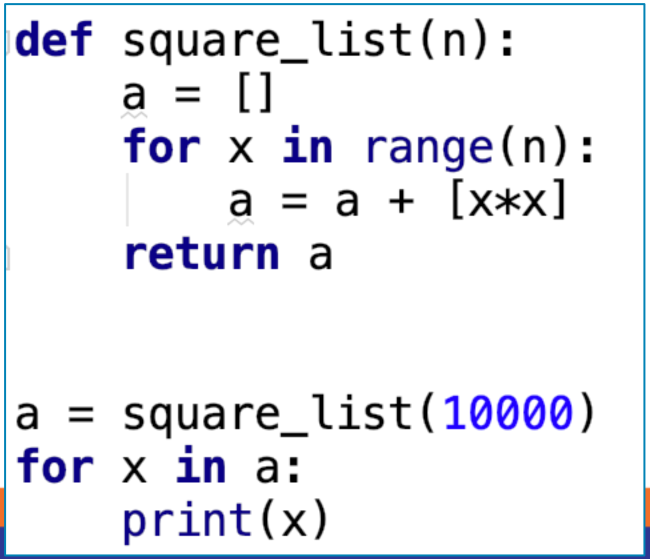
```python
def square_gen(n):
    for x in range(n):
        yield x*x


for x in square_gen(int(10000)):
    print(x)
```



```
0 1 4 9 16 25 36 49 64 81
```

- Generators are lazy. They only work on demand. That mean they can save cpu, memory, and other resources.

VS



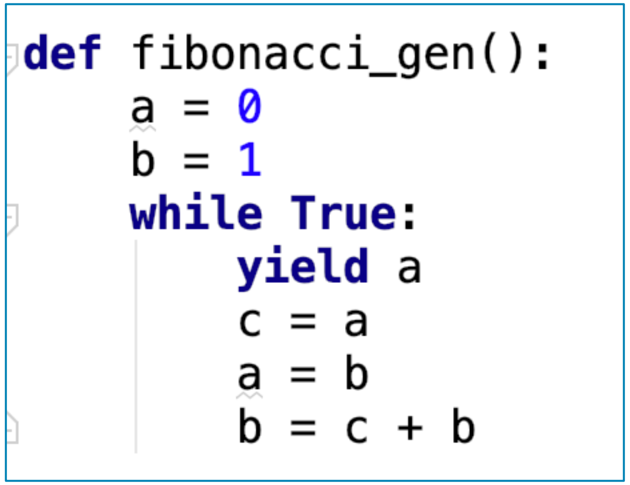
```python
def square_list(n):
    a = []
    for x in range(n):
        a = a + [x*x]
    return a


a = square_list(10000)
for x in a:
    print(x)
```

Generators vs normal functions

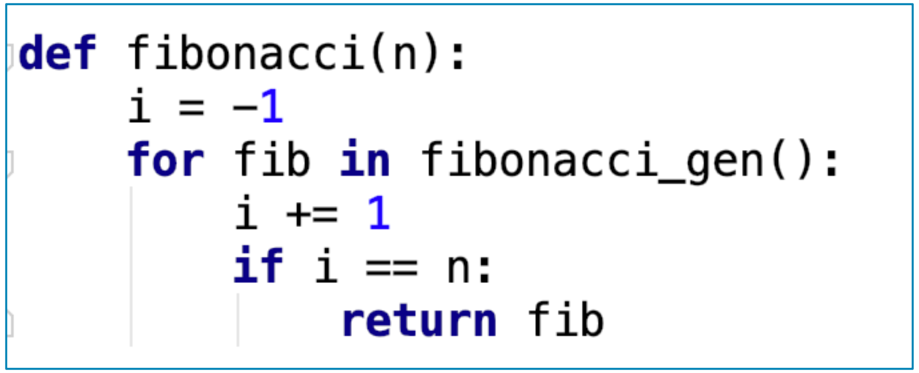
- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

Generate Fibonacci numbers

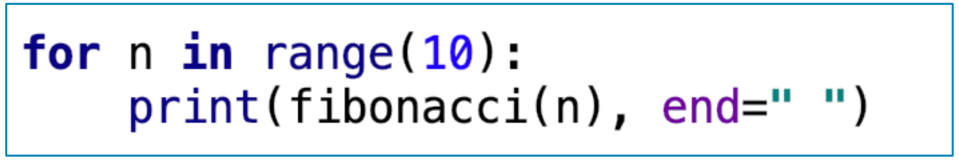
- Traditional example of generator



```python
def fibonacci_gen():
    a = 0
    b = 1
    while True:
        yield a
        c = a
        a = b
        b = c + b
```



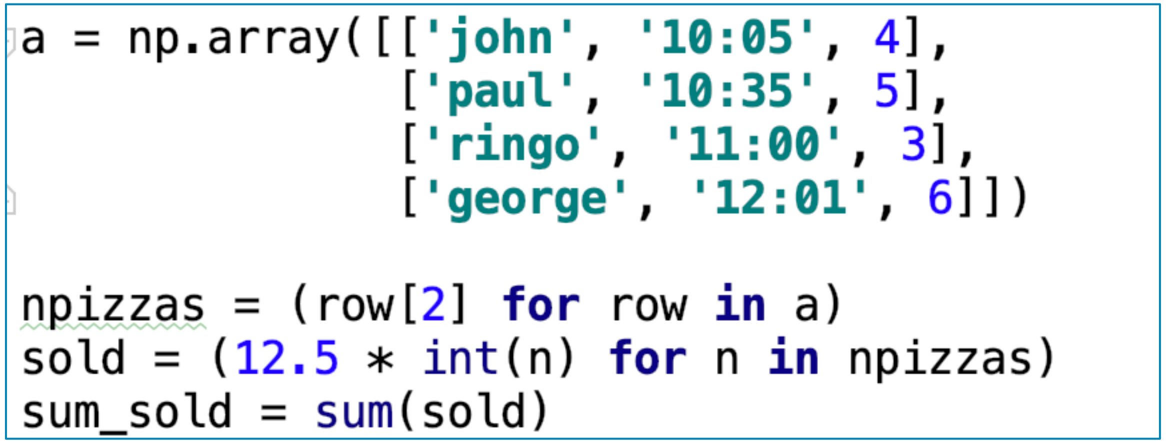
```python
def fibonacci(n):
    i = -1
    for fib in fibonacci_gen():
        i += 1
        if i == n:
            return fib
```



```python
for n in range(10):
    print(fibonacci(n), end=" ")
```

Pipelining generators

- Generators can be used successively

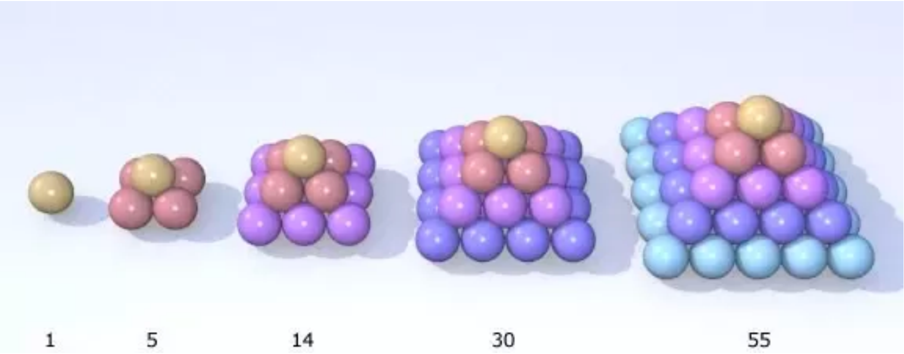


```
a = np.array([['john', '10:05', 4],
              ['paul', '10:35', 5],
              ['ringo', '11:00', 3],
              ['george', '12:01', 6]])

npizzas = (row[2] for row in a)
sold = (12.5 * int(n) for n in npizzas)
sum_sold = sum(sold)
```

```
Total sold: 225.000
```

exercises

1. Write a function that generate an odd number (giving n => 2n+1) by using generator with the formula: $a_n = a_{n-1} + 2$ ($a_0 = 1$)
2. Square pyramidal number counts the number of stacked balls in a square pyramid and can be calculated as $a_n = a_{n-1} + n^2$



Write a function that calculate pyramidal number by using generator

exercises

1. Write a function that can return each of the first three spherical Bessel functions

$$j_0(x) = \frac{\sin x}{x}$$

$$j_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x}$$

$$j_2(x) = \left(\frac{3}{x^2} - 1\right)\frac{\sin x}{x} - \frac{3 \cos x}{x^2}$$

a) Using normal function, take inputs as array x and the order n

b) Using generator for better performance