



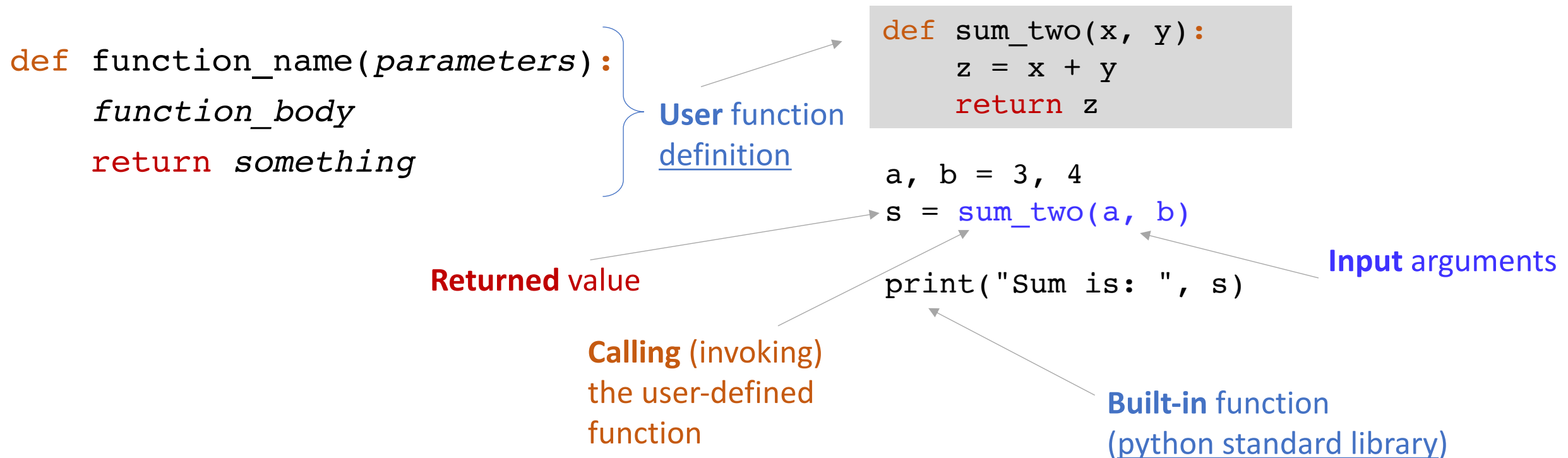
# 15-110 PRINCIPLES OF COMPUTING – F21

## LECTURE 14: FUNCTIONS

TEACHER:  
GIANNI A. DI CARO

# Functions: callable, named subprograms (procedures)

- **Function:** informally, a *subprogram*
  - we write a *sequence of statements* and give that sequence a *name*
  - the instructions can then be *executed at any point* in a program by referring to the *function name*



# All functions *return* something!

---

```
def function_name(parameters):  
    function_body
```

is implemented as:

```
def function_name(parameters):  
    function_body  
    return None
```



All function calls return something, `None` when left unspecified in the function body

# Built-in functions (Python standard library)

<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code><a href="#">bytearray()</a></code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

<https://docs.python.org/3/library/functions.html>

# Functions: organizing the code, putting aside functionalities

---

- Functions are a fundamental way to *organize* the code into **procedural elements** that can be *reused*
- Functions provide **structure and organization**, that facilitate:



# Decomposition – Abstraction - Reusability: an example

---

- ❖ A natural number  $n$  with  $D$  digits is named **armstrong** if the sum of  $D$ -th power of digits equals to  $n$  itself

Given  $n = d_1d_2d_3 \cdots d_D$       If  $d_1^D + d_2^D + d_3^D + \cdots + d_D^D = n$        $\rightarrow n$  is **armstrong**

- $n = 1450$     has  $D = 4$  digits:  $1^4 + 4^4 + 5^4 + 0^4 = 882 \rightarrow 1450$  is NOT armstrong
- $n = 153$     has  $D = 3$  digits:  $1^3 + 5^3 + 3^3 = 153 \rightarrow 153$  is armstrong!
- $n = 3$     has  $D = 1$  digit:  $3^1 = 3 \rightarrow 3$  is armstrong!

Write the function `is_special_number(n)` that takes as input a natural number  $n$  and returns `True` if the number is **either an armstrong or a prime number**, `False` otherwise.

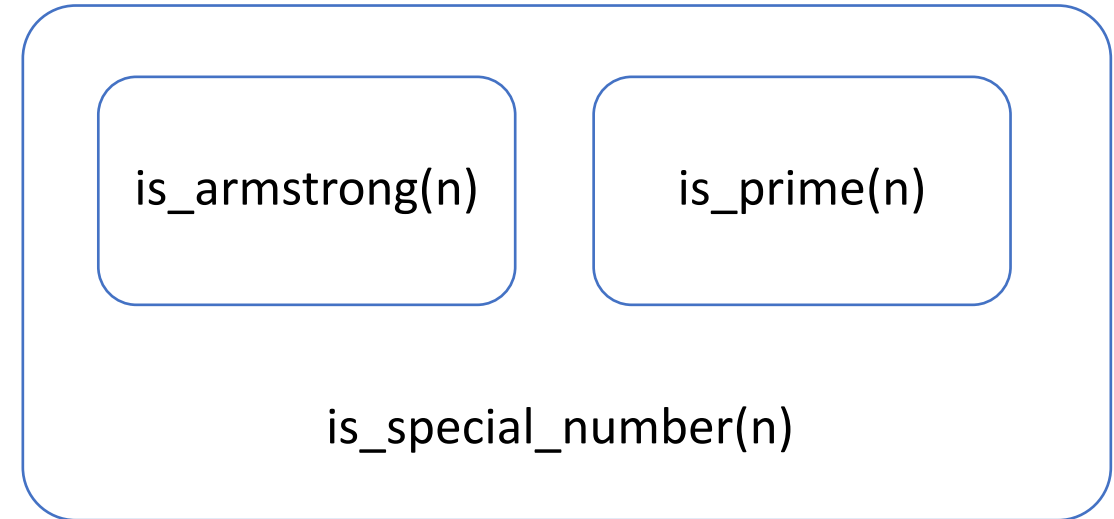
The function also prints out if the number is prime or armstrong.

# Decomposition – Abstraction - Reusability: an example

---

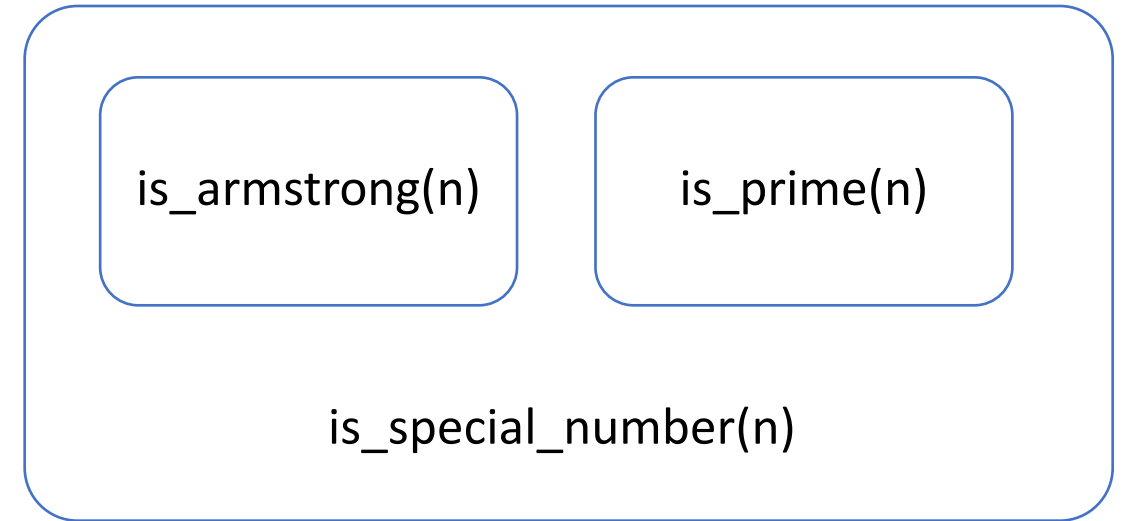
How do we design the function?

- **Decompose** the problem into sub-problems
  1. Check if it is armstrong or not  
→ Write the function `is_armstrong(n)`
  2. Check if it is prime or not  
→ Write the function `is_prime(n)`
  3. Combine the results of the checks and return the value



# Decomposition – Abstraction - Reusability: an example

```
def is_prime(n):  
    if n < 2:  
        return False  
    for i in range(2,n):  
        if n % i == 0:  
            return False  
    return True
```

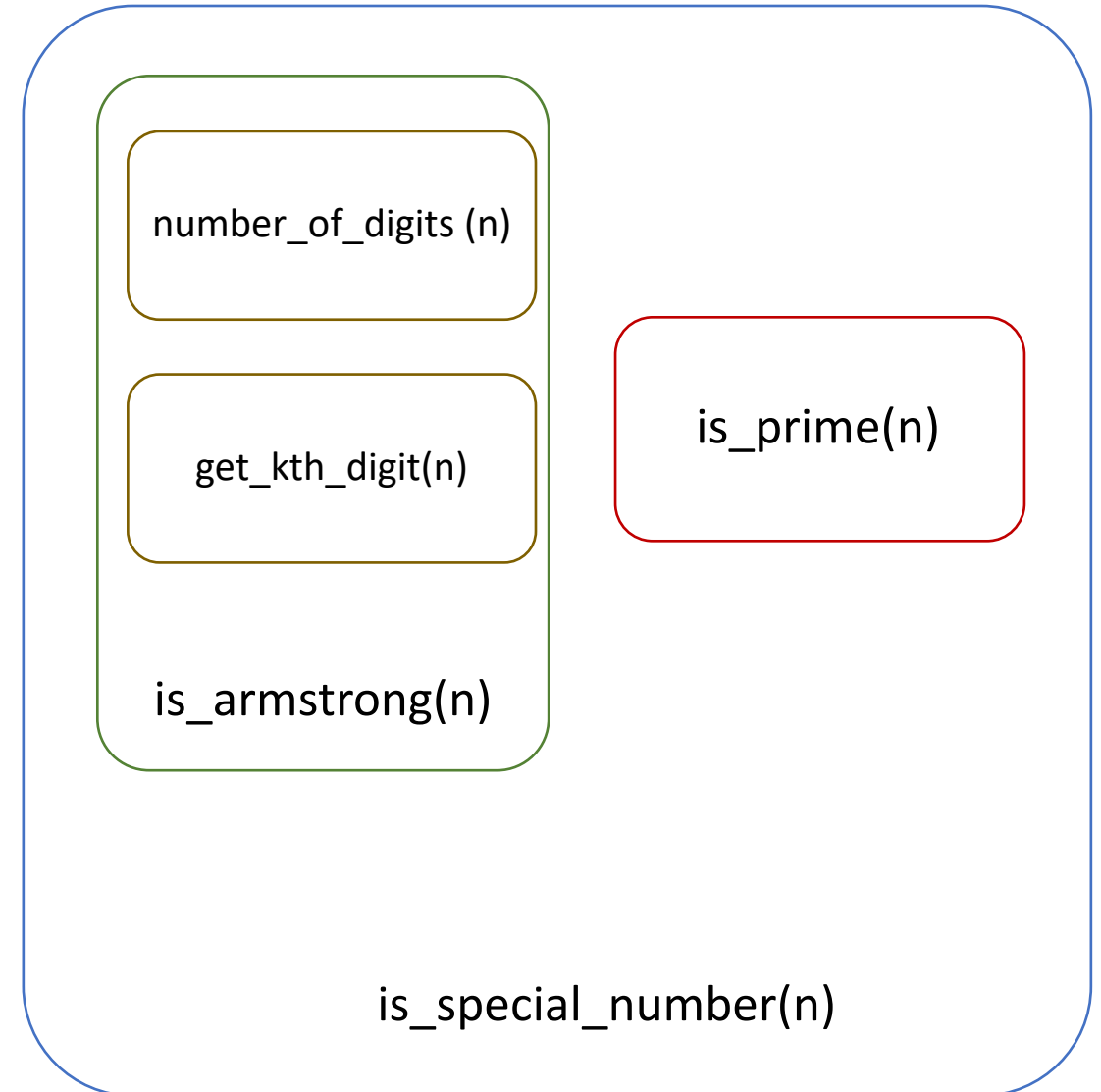


- What about the function `is_armstrong(n)`?
    - A natural number  $n$  with  $D$  digits is named **armstrong** if the sum of  $D$ -th power of digits equals to  $n$ 
      1. Get the **number of digits** of  $n$
      2. Get **each digit**, one by one
      3. Perform the **square and sum** operations
- What do we need to build the function?



# Decomposition – Abstraction - Reusability: an example

1. Get the **number of digits** of n  
→ Write the function `number_of_digits(n)`
2. Get **each digit**, one by one  
→ Write the function `get_kth_digit(n)`
3. Perform the **square and sum** operations



# Decomposition – Abstraction - Reusability: an example

---

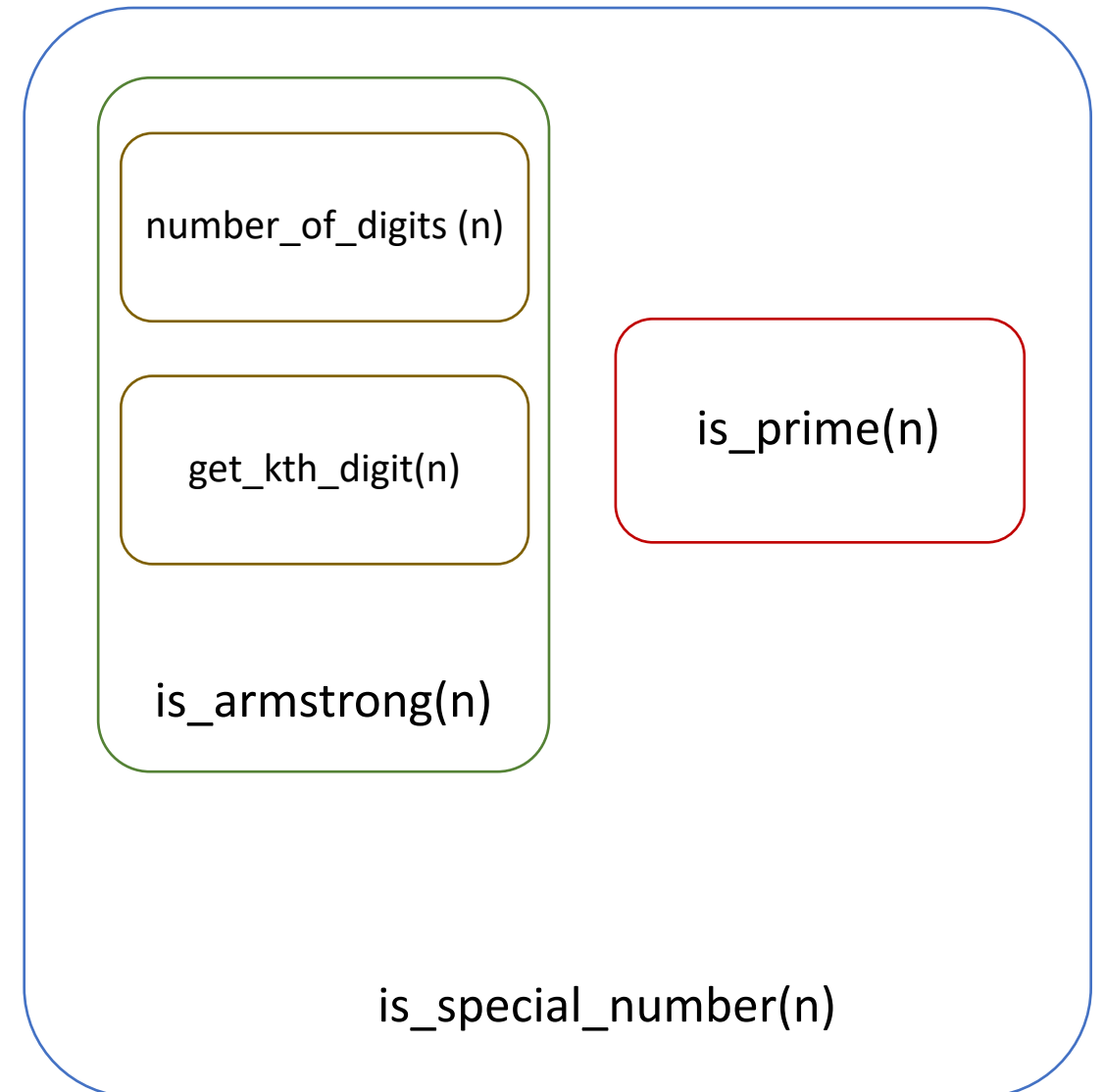
```
def number_of_digits(n):  
    count = 1  
    digit = n  
    while True:  
        digit //= 10  
        if digit == 0:  
            return count  
        else:  
            count += 1
```

```
def get_kth_digit(n, k):  
    n = abs(n)  
    n = n // 10 ** k  
    return n % 10
```

```
def is_armstrong(n):  
    n_digits = number_of_digits(n)  
    power_sum = 0  
    for k in range(n_digits):  
        digit = get_kth_digit(n, k)  
        power_sum += digit ** n_digits  
    if power_sum == n:  
        return True  
    else:  
        return False
```

# Decomposition – Abstraction - Reusability: an example

```
def is_special_number(n):  
    prime, armstrong = False, False  
    if is_prime(n):  
        print(n, 'is prime!')  
        prime = True  
  
    if is_armstrong(n):  
        print(n, 'is armstrong!')  
        armstrong = True  
  
    if prime or armstrong:  
        return True  
    else:  
        return False
```

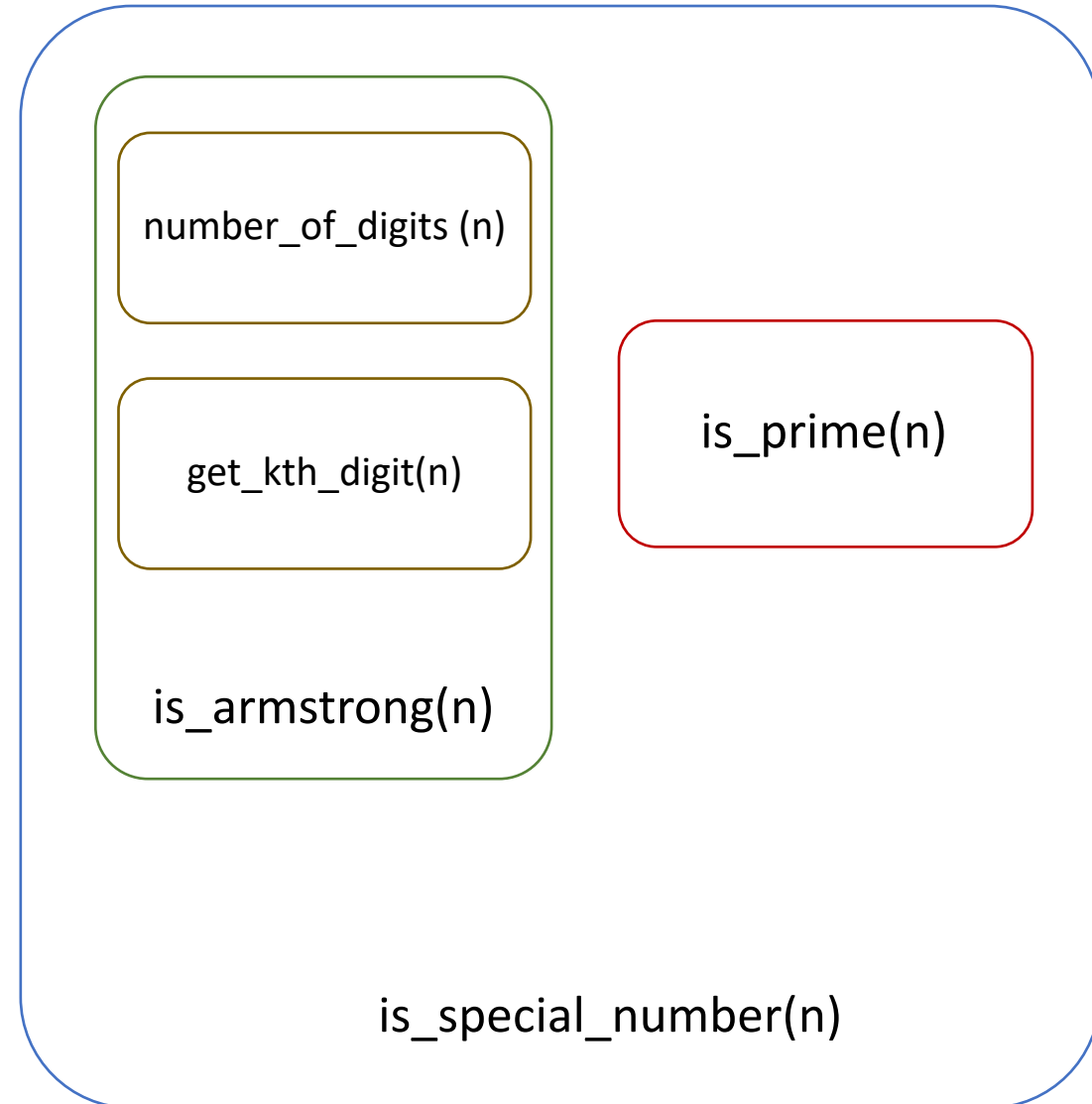


# Decomposition

➤ Instead of writing ONE function, we wrote FIVE functions!

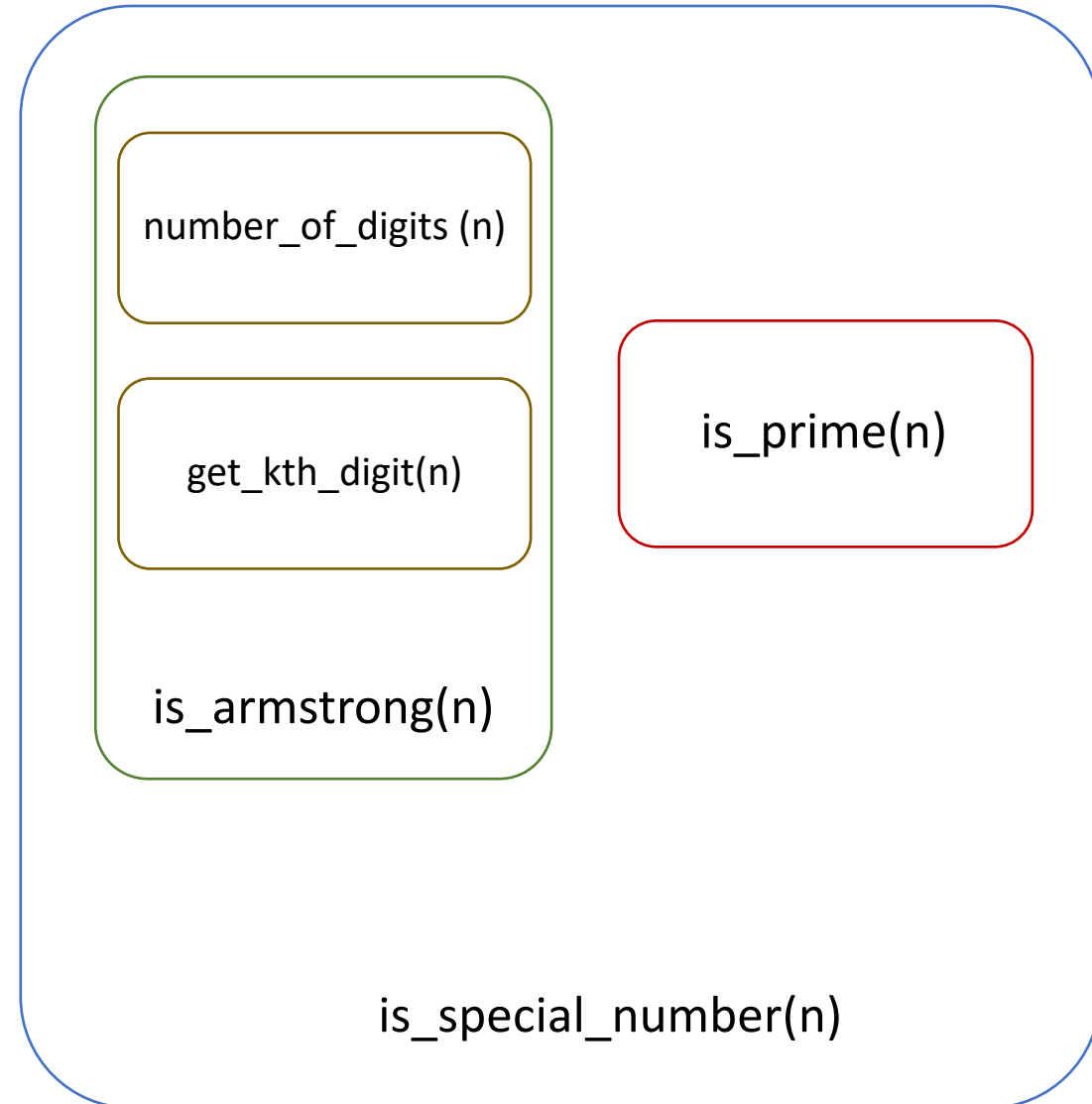
❖ **Problem decomposition:** we solved four sub-problems and combined them to tackle the target problem

- ✓ Each sub-problem was easier to tackle than the whole, original problem
- ✓ Each sub-problem is relatively easy to debug and test for correctness and efficiency standalone
- ✓ We gained in readability of the whole solution: we can actually *read* the code of `is_special_number(n)`
- ✓ We can split the job in parallel among multiple programmers, each doing a function! ;-)



# Reusability

- ❖ **Reusability:** we have now five functions (working, well debugged!) each doing a different thing (i.e., providing a different service)
  - ✓ We can **reuse the individual functions in different, new problems!**
    - E.g., find all the happy prime numbers up to n
  - ✓ We can reuse existing functions **to define we functions** with improved / extended functionalities
    - E.g., find the number of digits that are even/odd



# Reusability

---

## ❖ Reusability:

- ✓ We can **reuse the same function as many times as we want in our code**, without having to repeat the code, but just invoking the function by its name

```
def is_armstrong(n):  
    n_digits = number_of_digits(n)  
    power_sum = 0  
    for k in range(n_digits):  
        digit = get_kth_digit(n, k)  
        power_sum += digit ** n_digits  
    if power_sum == n:  
        return True  
    else:  
        return False
```

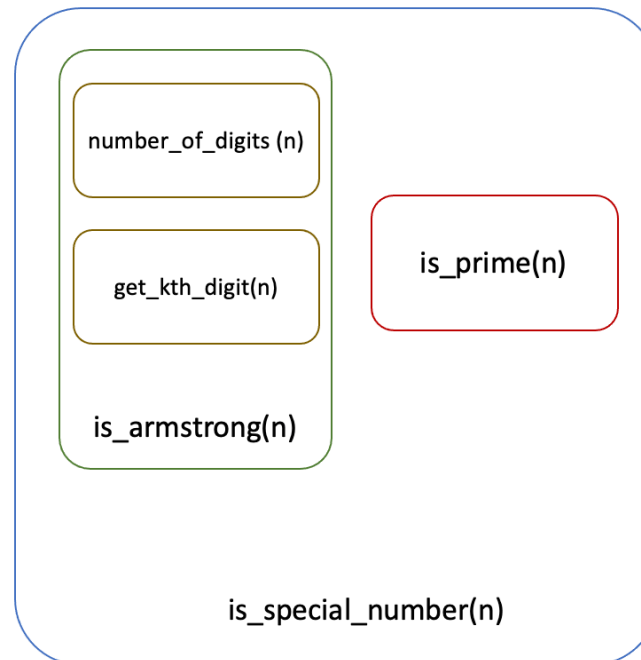
We call the `get_kth_digit()` function `n_digits` times!

```
def happy():  
    print("Happy birthday to you!")
```

```
def sing(person):  
    happy()  
    happy()  
    print "Happy Birthday, dear", person)  
    happy()
```

# Abstraction

- ❖ **Abstraction:** functions provides a service **hiding the details about how the service is being provided**. We can use a function by **invoking its name** and using its **results!**
  - ✓ Solution details are **abstracted away** (to the user)
  - ✓ A function is not limited to a specific input, instead it describes a **computation** that applies to **any admissible input** (any natural number  $n$  in this case) since the input is **parametric**



# Abstraction

---

```
def number_of_digits(n):  
    count = 1  
    digit = n  
    while True:  
        digit //= 10  
        if digit == 0:  
            return count  
        else:  
            count += 1
```

```
def number_of_digits(n):  
    return len(str(n))
```

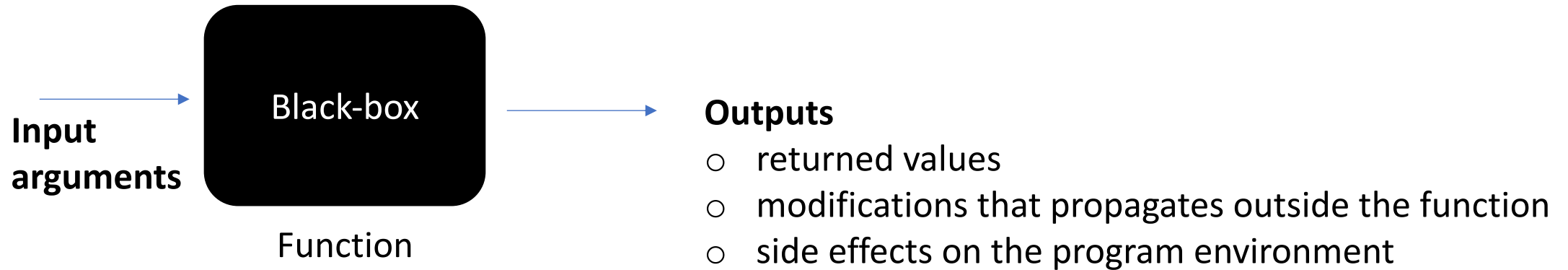
Three different ways of implementing the same functionality (i.e., returning the same values for the same inputs)

```
import math  
def number_of_digits(n):  
    ...  
    n: integer > 0  
    Observe: log10(1) is 0, log10(10) is 1, log10(100) is 2,  
    log10(1000) is 3 ...  
    ...  
    return math.floor( math.log10(n)) + 1
```



# Abstraction using functions

---



- The only information relevant to use a function :
  - **input parameters**: types and admitted values
  - **returned objects**: types, range of values, adopted conventions
  - **description of what the function does**, including possible side effects

The precise details about *how* processing is performed are **hidden by the abstraction**

# Local and global scope of variables

❖ **Local Scoping**: where do function variables *live*?

```
def average(L):  
    n_elements = len(L)  
    avg = sum(L) / n_elements  
    return avg
```

```
numbers = [1, 3, 6, 2]  
x = average(numbers)  
print(x)
```

```
print(avg) ?
```

**NameError**: name 'avg' is not defined

```
print(n_elements) ?
```

```
print(L) ?
```


- Variables `avg`, `n_elements`, `L` are **local to the function**
- Their **scope is limited to the function and to the duration of the function call**
- ❖ Variable defined inside a function **cannot be referenced outside the function!**

# Local and global scope of variables

❖ **Global Scoping**: where do variables *live*?

✓ This works, variable `L` is defined outside of a function and as such can be accessed **globally**

```
def average_global():  
    n_elements = len(L)  
    avg = sum(L) / n_elements  
    return avg
```



```
L = [1, 3, 6, 2]  
x = average_global()  
print(x)
```

➤ `L` is a variable that can be **referred to anywhere**  
→ Variable with a *global* scope

```
numbers = [1, 3, 6, 2]  
x = average_global()
```

**NameError**: name 'L' is not defined

○ This doesn't work, `L` isn't defined anywhere

# Local and global scope of variables

---

❖ **Global Scoping**: *clean* way to define global variables

```
def average_global():  
    global L  
    n_elements = len(L)  
    avg = sum(L) / n_elements  
    return avg
```

```
L = [1, 3, 6, 2]  
x = average_global()  
print(x)
```

➤ In the function `L` is explicitly defined as a **global variable**: we assume that it has been **declared somewhere**

# Local and global scope of variables

---

❖ **Global Scoping**: changes are propagated outside of the function!

```
def median():  
    global L  
    L.sort()  
    return L[ len(L)//2 ]
```

```
L = [1, 3, 6, 2, 0]  
m = median()  
  
print(L)
```

L has changed now: [0, 1, 2, 3, 6]

➤ Be careful: **Avoid** as much as possible to use global variables!

# Functions without input parameters

---

```
def function_name():  
    function_body  
    return something
```

The function **does something** and **doesn't require any input specifications** for doing it!

```
def happy():  
    print("Happy birthday to you!")
```

```
def product():  
    global L  
    prod = 1  
    for v in L:  
        prod = prod * v  
    return prod
```

```
def square():  
    sizeh = 22  
    sizev = 10  
    for i in range(sizeh):  
        print('-', end = '')  
    print()  
    for i in range(sizev):  
        print('|', end = '')  
        for j in range(sizeh-2):  
            print(' ', end = '')  
        print('|', end = '')  
        print()  
    for i in range(sizeh):  
        print('-', end = '')  
    print('\n')
```

# Functions with input parameters

```
def function_name(parameters):  
    function_body  
    return something
```

The function **does something** and **does require** input data / specifications for doing it!

- ✓ If we only need to pass one **single argument**, there's no much to say
  - The situation is different when passing **multiple parameters** ...

```
def quadratic_roots(a, b, c):  
    d = 1 / (2 * a)  
    sqr = math.sqrt(b**2 - (4 * a * c))  
    return (-b + sqr) * d, (-b - sqr) * d
```

$$ax^2 + bx + c = 0$$

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

`quadratic_roots(31, 93, 62)` is different than: `quadratic_roots(93, 31, 62)`

**Order matters!**

# Functions with input parameters: use of *named* arguments

---

```
def quadratic_roots(a, b, c):  
    d = 1 / (2 * a)  
    sqr = math.sqrt(b**2 - (4 * a * c))  
    return (-b + sqr) * d, (-b - sqr) * d
```

- ✓ Passing arguments as **positional arguments**

```
quadratic_roots(31, 93, 62)
```

is different than:

```
quadratic_roots(93, 31, 62)
```

**Order matters!**

- ✓ Passing arguments as **keyword arguments**

```
quadratic_roots(a=31, b=93, c=62)
```

is the same as:

```
quadratic_roots(b=93, a=31, c=62)
```

**Order doesn't matter!**



# Keyword arguments and the `help()` function

---

- Passing arguments as **keyword arguments** works because python **knows the name function arguments**, and therefore it can perform **automatic matching without errors**

→ We can ask python **help** on function's parameters using the `help(function_name)`:

`help(quadratic_roots)` gives as answer: `quadratic_roots(a,b,c)`

- ✓ Use of keyword arguments also increases the **clarity of a program!**

`random_password(upper=1, lower=1, digits=1, length=8)`

vs.

`random_password(1, 1, 1, 8)`

# Default values for the parameters, equivalent function calls

- When defining a function, a **default value can be defined for each argument**
  - If a value is passed for a parameter with a default value when the function is called, then the **parameters takes the value of the provided argument**
  - Otherwise, the parameter takes its **default value**

```
def quadratic_roots(a = 1, b = -3, c = 2):  
    d = 1 / (2 * a)  
    sqr = math.sqrt(b**2 - (4 * a * c))  
    return (-b + sqr) * d, (-b - sqr) * d
```

Default equation:

$$x^2 - 3x + 2 = 0$$

```
x1, x2 = quadratic_roots()
```

```
x1 → 2.0
```

```
x2 → 1.0
```

```
x1, x2 = quadratic_roots(a=2, b=3, c=1)
```

```
x1 → -0.5
```

```
x2 → -1.0
```

# Default values for the parameters, equivalent function calls

---

```
def quadratic_roots(a = 1, b = -3, c = 2):  
    d = 1 / (2 * a)  
    sqr = math.sqrt(b**2 - (4 * a * c))  
    return (-b + sqr) * d, (-b - sqr) * d
```

`x1, x2 = quadratic_roots(b=2, c=-3)`      Equivalent to call the function as:

`x1` → 1.0  
`x2` → -3.0

`quadratic_roots(a=1, b=2, c=-3)`

`x1, x2 = quadratic_roots(b=4)`      Equivalent to call the function as:

`x1` → -0.58  
`x2` → -3.41

`quadratic_roots(a=1, b=4, c=2)`

# Be careful mixing up positional and named parameters!

---

```
def quadratic_roots(a = 1, b = -3, c = 2):  
    d = 1 / (2 * a)  
    sqr = math.sqrt(b**2 - (4 * a * c))  
    return (-b + sqr) * d, (-b - sqr) * d
```

```
x1, x2 = quadratic_roots(2, c=-3)
```

Equivalent to call the function as:

```
x1 → 2.18  
x2 → -0.68
```

```
quadratic_roots(a=2, b=-3, c=-3)
```

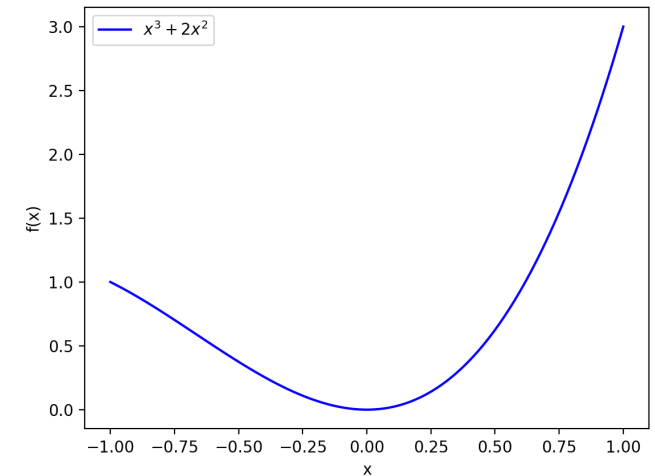
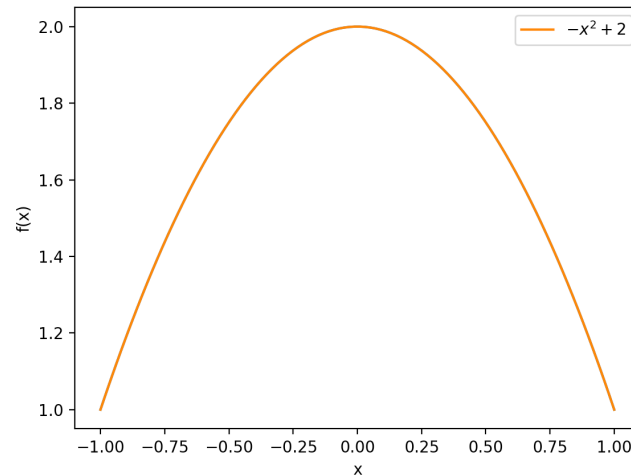
```
x1, x2 = quadratic_roots(2, a=3) TypeError: quadratic_roots() got multiple values for argument 'a'
```

```
x1, x2 = quadratic_roots(b=2, 3, 4) SyntaxError: positional argument follows keyword argument
```

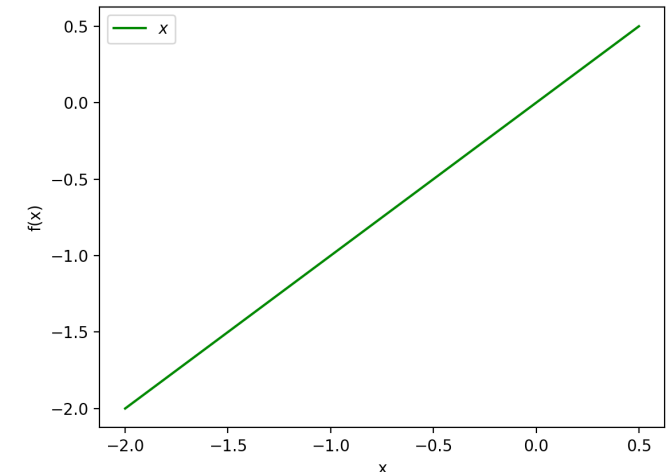
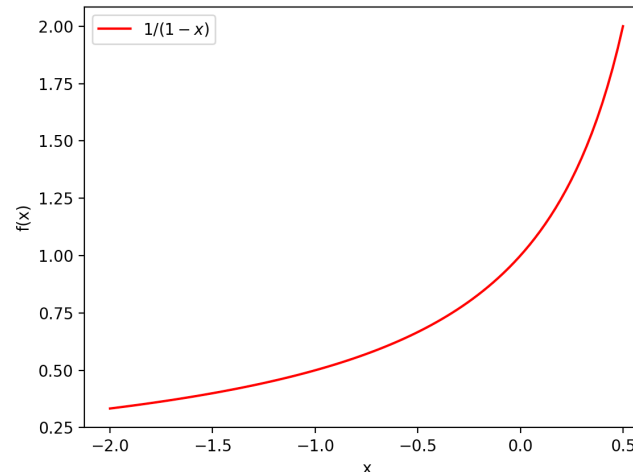
# Passing *functions* as arguments!

- Function parameters can also include a **function**, that can be then invoked inside the function itself

```
def parabola(x):  
    return -x**2 + 2  
  
def cubic(x):  
    return x**3 + 2*x**2
```



```
def geometric(x):  
    return 1 / (1-x)  
  
def line(x):  
    return x
```



# Passing functions as arguments!

---

- Function parameters can also include a **function**, that can be then invoked inside the function itself

```
def estimate_max_in_interval(f,
                             xmin, xmax,
                             samples):
    x = xmin
    step = (xmax - xmin) / samples
    max_val = f(xmin)
    for i in range(samples):
        y = f(x)
        if y > max_val:
            max_val = y
        x += step
    return max_val
```

```
def parabola(x):
    return -x*x + 2

def cubic(x):
    return x*x*x + 2*x*x

def geometric(x):
    return 1 / (1-x)

def line(x):
    return x
```

```
print(estimate_max_in_interval(parabola, -1, 1, 100))
```

# Function specifications, docstrings, help()

- The only information relevant to use a function :
  - ✓ **input parameters**: types and admitted values
  - ✓ **returned objects**: types, range of values, adopted conventions
  - ✓ **description of what the function does**, including possible side effects

*Where do we need to write/store this information?*

```
def average(L):  
    '''  
    Parameters  
    -----  
    L : List of numbers  
  
    Returns  
    -----  
    avg : float, the average of the input list  
    '''  
    n_elements = len(L)  
    avg = sum(L) / n_elements  
    return avg
```

- If the first line after the name of the function is a string delimited by triple quotes → **docstring**

A docstring shall contain the description of the I/O of the function and what the function does

# Function specifications, docstrings, help()

```
def average(L):  
    '''  
    Parameters  
    -----  
    L : List of numbers  
  
    Returns  
    -----  
    avg : float, the average of the input list  
    '''  
    n_elements = len(L)  
    avg = sum(L) / n_elements  
    return avg
```

- ✓ If you don't remember about a function or want to know about it: [ask python!](#)
- The docstring is returned by the `help( )` function:

```
help(average)
```

```
Help on function average in module __main__:
```

```
average(L)  
    Parameters  
    -----  
    L : List of numbers  
  
    Returns  
    -----  
    avg : float, the average of the input list
```



# Function specifications, docstrings, help()

---

- ✓ Can ask help about any function!

`help(sum)`

Help on built-in function sum in module builtins:

`sum(iterable, start=0, /)`

Return the sum of a 'start' value (default: 0) plus an iterable of numbers

When the iterable is empty, return the start value.

This function is intended specifically for use with numeric values and may reject non-numeric types.

# Specifications

---

A specification **describes the abstraction**, the function and its elements in our case, in order to properly use it and reuse it!

- **Specification:** defines a *contract* between the provider/implementer of an abstraction (a function) and those who will be using the abstraction (the function), the users
- **User** ↔ **Client** for the services provided by the abstraction (the function)



# Specifications: example

**Assumptions:** what the client must do to use the function

**Guarantees:** what the provider guarantees about outputs and effects of the function

**Numeric examples:** show function usage and provide basic test cases

```
def find_root(f, a, b, N):  
    '''Approximate solution of f(x)=0 on interval [a,b] by the bisection method.  
  
    Parameters  
    -----  
    f : function  
        The function for which we are trying to approximate a solution f(x)=0.  
    a,b : numbers  
        The interval in which to search for a solution. The function returns  
        None if f(a)*f(b) >= 0 since a solution is not guaranteed.  
    N : (positive) integer  
        The number of iterations to implement.  
  
    Returns  
    -----  
    x_N : number  
        The midpoint of the Nth interval computed by the bisection method. The  
        initial interval [a_0,b_0] is given by [a,b]. If f(m_n) == 0 for some  
        midpoint m_n = (a_n + b_n)/2, then the function returns this solution.  
        If all signs of values f(a_n), f(b_n) and f(m_n) are the same at any  
        iteration, the bisection method fails and return None.  
  
    Examples  
    -----  
    >>> f = lambda x: x**2 - x - 1  
    >>> bisection(f,1,2,25)  
    1.618033990263939  
    >>> f = lambda x: (2*x - 1)*(x - 3)  
    >>> bisection(f,0,1,10)  
    0.5  
    '''
```

... the program code would follow