Understanding functions

formally and informally

Sukrit Gupta

January 17, 2024

Acknowledgement and disclaimer

All mistakes (if any) are mine.
I have used several sources which I have referred to in the appropriate places.

Outline

Section 1

The intuition behind functions

Functions around us ...

Let go of all your preconceived notions about the topic. Follow me.

Imagine that I ask you to differentiate between the following two vehicles using the given images:

Figure: Hyundai Santro and Hyundai Verna (credit: Autocar India)

What factors do you consider for this?

Let's say: Color, Weight, Length?

Being aware of the process

What we are trying to do? From the given image, understand how the two cars look like. Let's try to summarize the process:

- Using visual aids (our eyes), we see the two images. (Input: Images of vehicles.)
- Our brain determines important differentiating factors for the two vehicles. (Features = ?)
- We assign certain value of the features to the two vehicles. For example: if feature color = red, then vehicle = Verna.

Formalizing it

- What we have done is, we have involuntarily, subconsciously learnt a function that maps the images and cars.
- Formally:

$$f : \mathcal{X} \rightarrow \mathcal{Y}$$

  where $\mathcal{X}$ is the set of input values, and $\mathcal{Y}$ is the set of output values.
- Can anyone tell me what $\mathcal{X}$ and $\mathcal{Y}$ are in our case?
- In our case, $\mathcal{X} = \{(\text{silver}, 900kg, 3600mm), (\text{red}, 1100kg, 4440mm)\}$ and $\mathcal{Y} = \{\text{Santro}, \text{Verna}\}$
- So, $f(\text{silver}, 900kg, 3600mm) = \text{Santro}$ and $f(\text{red}, 1100kg, 4440mm) = \text{Verna}$

Section 2

Formal definitions

Functions

Definition

A function is a rule that assigns each input *exactly one* output. So what output do you assign to an input, x, needs to be determined by a single rule.

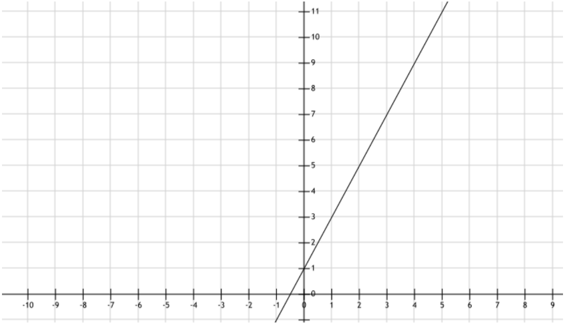


Figure: Function $f(x) = 2x + 1$.

Parts of a Function

- **Domain**: The set of inputs, $\mathcal{X}$.
- **Codomain**: The set of *allowable* outputs.
- **Range**: The actual set of outputs of a function.
- **Mapping**: The description of output value $y \in \mathcal{Y}$ in terms of input value $x \in \mathcal{X}$.
- We would write $f : \mathcal{X} \to \mathcal{Y}$ to describe a function with name f, domain $\mathcal{X}$ and codomain $\mathcal{Y}$.

A closer look at functions

For example, consider the function $f : \mathbb{R} \to \mathbb{R}, f(x) = 0.1x^2 + 5$.

What is the domain and codomain of this function?

What will this function look like? Any guesses?

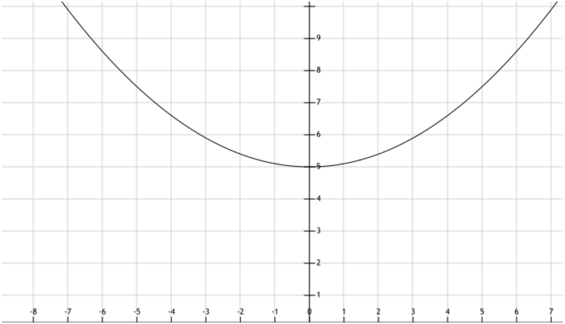


Figure: $f(x) = 0.1x^2 + 5$.

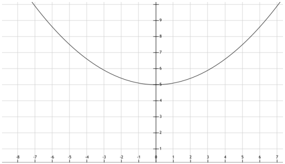


Figure: $f(x) = 0.1x^2 + 5$.

$f(1) = 0.1 \times 1 + 5$
$f(2) = 0.1 \times 4 + 5$
$f(3) = 0.1 \times 9 + 5$

- Same domain and codomain?
- Same codomain and range?
- Not every real number actually is an output (there is no way to get values $\in [0, 5)$)

By the way ...

Look at this classifier that distinguishes between images of digits. What does it do?

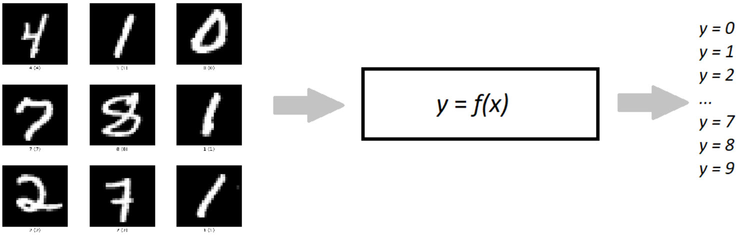


Figure: Classifying hand-written digits.

Section 3

Types of functions

Surjective/Onto functions

- When range equals codomain, the function is called surjective or onto.
- The terminology should make sense: the function puts the range (entirely) *on top* of the codomain.
- Is $f : \mathbb{R} \to \mathbb{R}, f : x \to x^2$ an onto function?
- What about $f : \mathbb{R} \to [0, \infty), f : x \to x^2$?

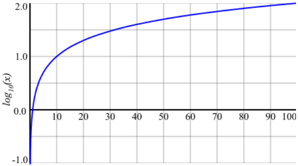
Injective/One-to-one functions

- When each element of the codomain is mapped to at most one element of the domain, the function is called one-to-one or injective.
- Is $f : \mathbb{R} \to \mathbb{R}, f : x \to x^2$ a one-to-one function?
- What about $f : \mathbb{R} \to \mathbb{R}, f : x \to x + 1$?

Bijective/Both Surjective and Injective

- It should be clear that there are functions which are surjective, injective, both, or neither.
- A function that is both one-to-one and onto (an injection and surjection), is called a bijective function.
- Is $f : \mathbb{R} \to \mathbb{R}, f : x \to x + 1$ a bijective function?

Q: What type of fn. is $f : \mathbb{R}^+ \rightarrow \mathbb{R}, f : x \rightarrow log(x)$



Why know about different types of functions?

- When discussing functions, we go from element in the domain (say x) to its corresponding element in the codomain (denoted by $f(x)$).
- What if we want to go the other way?
- Start with an element from the codomain (say y) and understand which element or elements (if any) from the domain it is mapped to.
- Suppose $f : \mathcal{X} \to \mathcal{Y}$, for $y \in \mathcal{Y}$, $f^{-1}(y)$ represents the set of all elements in the domain X that are mapped to y. That is, $f^{-1}(y) = \{x \in \mathcal{X}, f(x) = y\}$.
- Although $f^{-1}(y)$ is defined (but may not exist) for any function f, inverse functions only exist for bijections.

Section 4

Real world functions

Why do you need functions in Python

- Imagine you have a car.
- The car has multiple components.
- All the components coordinate with each other and also work independently.
- Imagine that your car has an issue and you take it to a mechanic.
- The mechanic does not open all the components in your car, but instead tries to locate the specific component that has an issue.
- This is possible because the car has a modular design.

Why functions?

- Real-world systems are modular.
- Functions help reduce the amount of code. The more code a program contains, the more chance there is for something to go wrong, and the harder the code is to maintain.
- Also functions provide a level of abstraction.

Q: Can you think of more modular systems?

- Uber mobile application.
- ATM machine.
- Mobile phone.

Section 5

Functions in Python

Syntax

```
def name_of_function (list of formal parameters):
    body of function
```

`def` is a reserved word that tells Python that a function is about to be defined.

The function name is simply a name that is used to refer to the function.

Example

Suppose you have the function: $y = f(x) = x^2$. How to code it?

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

y = f(3)
```

Example

```python
def max_(x, y):
    if x > y:
        return x
    else:
        return y

z = max_(3, 4)
```

- x, y in the function definition are the formal parameters of the function.
- During function call the formal parameters are bound to the actual parameters (or arguments) of the function call. For example, the function call max_(3, 4) binds x to 3 and y to 4.

Function call

To recapitulate, when a function is called:

1. The actual parameters are evaluated, and the formal parameters of the function are bound to the resulting values.
2. The point of execution (the next instruction to be executed) moves from the point of invocation to the first statement in the body of the function.
3. The code in the body of the function is executed until either a return statement is encountered (value of function invocation is the value of the expression following the return) or there are no more statements to execute (**function returns the value `None`**).
4. The value of the invocation is the returned value.
5. The point of execution is transferred back to the code immediately following the invocation.

Functions with/without a return statement

Take two examples of functions:

```
def max1(x, y):
    if x > y:
        return x
    else:
        return y

def max2(x, y):
    if x > y:
        print(x)
    else:
        print(y)

def max3(x, y):
    if x > y:
        print(x)
    else:
        print(y)
    return None
```

Section 6

Variable Scope

Variable scope

```
def f(x): #name x used as formal parameter
    y = 1
    x = x + y
    print ('x =', x) #
    print ('y =', y) #
    return x
x = 3
y = 2
z = f(x) #value of x used as actual parameter
print ('z =', z)
print ('x =', x) #
print ('y =', y) #
```

- Each function defines a new name space, also called a scope.
- The formal parameter `x` and local variable `y` used in `f` exist only within the scope of f. Assignments in `f` have no effect on the bindings of the names `x` and `y` that exist outside the scope of `f`.

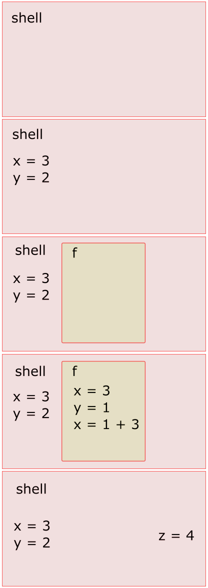
How to think about variable scope

Here's one way to think about this:

- At top level, i.e., the level of the shell, a symbol table keeps track of all names defined at that level and their current bindings.
- When a function is called, a new symbol table (sometimes called a stack frame) is created. This table keeps track of all names defined within the function (including the formal parameters) and their current bindings.
- If a function is called from within the function body, yet another stack frame is created.
- When the function completes, its stack frame goes away.

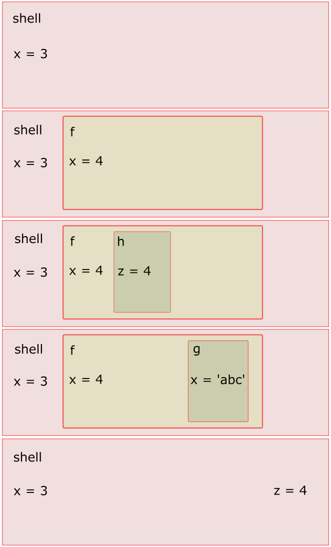
How to think about variable scope

```
def f(x):
    y = 1
    x = x + y
    print ('x =', x) #
    print ('y =', y) #
    return x
x = 3
y = 2
z = f(x)
print ('z =', z)
print ('x =', x) #
print ('y =', y) #
```



Q: Draw symbol table.

```
def f(x):
    def h():
        z = x
        print ('z =', z) #
    def g():
        x = 'abc'
        print ('x =', x) #
    x = x + 1
    print ('x =', x) #
    h()
    g()
    print ('x =', x) #
    return x
x = 3
z = f(x)
print ('x =', x) #
print ('z =', z) #
```



What did we learn today?

1. The intuition behind functions
2. Formal definitions
3. Types of functions
4. Real world functions
5. Functions in Python
6. Variable Scope

Thank you!