Extending default vocabulary with functions

Lecture 03.01

By *Marina Barsky*

What operators (verbs, actions) do we already know?

We can have extended operations on variables and values with *functions*:

```
max_number = max (1,4)
min_in_list = min ([1.5, 5, 10.6])
max_char = max ('Hello world')
```

What functions exists?

```
>>> dir(__builtins__)
```

What does each function do?

```
>>> help (max)
>>> help (round)
```

Functions are evaluated from inside out

```
a = -14
b = 3
c = -4
d = 11
absolute_max = max(abs(a), abs(b), abs(c), abs(d))


min_of_max = min(max(a,b), max(c,d))


max_sum= max(sum(a, b), sum(c, d))
```

Sorting function

```
a = [12, 10, 21]
b = sorted(a)

s = 'bath'
t = sorted(s)
```

Enso – abstract symbol for absolute enlightenment: strength, elegance, the universe, and mu (the void)

Abstraction

- Abstraction is a way of managing complexity
- Each piece of program is an abstraction
- Abstraction contains only important features
- Abstraction hides (encapsulates) complexity and makes your story easier to read

We use abstraction in our language all the time

Recipe:

How?

What is that?

- *Preheat oven* to *320^{o} Fahrenheit*
- …

What is oven?

The structured programming paradigm: daily activities example

Split this list into three blocks of related activities and give each block a heading

- Get out of bed
- Eat breakfast
- Park the car
- Get dressed
- Get the car out of the garage
- Drive to work
- Find out what your boss wants you to do today
- Feedback to the boss on today's results.
- Do what the boss wants you to do

Daily activities: 3 sub-programs

Get up

- Get out of bed
- Get dressed
- Eat breakfast

Go to work

- Get the car out of the garage
- Drive to work
- Park the car

Do your job

- Find out what your boss wants you to do today
- Feedback to the boss on today's results.
- Do what the boss wants you to do

We can work on each part separately

Improve instructions in **go to work** module:

- Listen to the local traffic and weather report
- Decide whether to go by bus or by car
- If going by car, get the car and drive to work.
- Else walk to the bus station and catch the bus

Other modules are not affected

History: the progress of abstraction I

- The 1950s – *Machine code* is common. Assembly language abstracts an underlying computing machine
- The 1960s - "Imperative" languages (FORTRAN, BASIC, C) – built as an abstraction level upon Assembly. Their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve.

History: the progress of abstraction II

- The 1970s - The alternative to modeling the machine is to model the problem you're trying to solve.
- Early languages such as LISP and APL chose particular views of the world ("All problems are ultimately lists" or "All problems are algorithmic," respectively).
- PROLOG casts all problems into chains of decisions.

History: the progress of abstraction III

- The 1980s – “modular” languages (Modula-2, ADA) – can work on each part separately, precursors of modern Object-oriented languages – Java, Python

defining our own abstractions and teaching machines how to interpret them

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

def greeting (name):
    print ('Happy birthday, dear ' + name)
```

We can now rewrite the song using new abstractions

```
#now in the program we can call these functions
happy()
happy()
greeting ('Peter')
happy()
```

This is an example of using functions as **sub-programs (sub-routines)** to make set of instructions *modular*

How functions look: syntax

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

Reserved word **def**

```
def greeting (name):
    print ('Happy birthday, dear ' + name)
```

How functions look: syntax

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

Function **name**

```
def greeting (name):
    print ('Happy birthday, dear ' + name)
```

How functions look: syntax

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

Function
parameters

```
def greeting (name):
    print ('Happy birthday, dear ' + name)
```

How functions look: syntax

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

Function **body (indented)**

```
def greeting (name):
    print ('Happy birthday, dear ' + name)
```

All functions should have a docstring

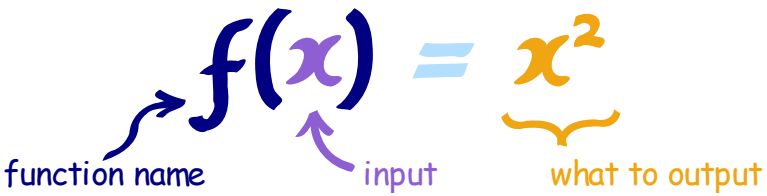
```
def greeting (name):
    """ (str) -> None

    Prints customized greeting by
        concatenating name with the constant

    >>> greeting ('Ann')
    'Happy birthday, dear Ann'
    """

    print('Happy birthday, dear ' + name)
```

Functions as mathematical concepts (proper functions)



parameter

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

return value

```
result = f(4)
```

calling function f with argument 4

Function design recipe (6 steps)

1. **Examples**
   - What should your function do?
   - Type a couple of example calls.
   - Pick a name (often a verb or verb phrase)
2. **Type Contract**
   - What are the parameter types?
   - What type of value is returned?
3. **Header**
   - Pick meaningful parameter names.
4. **Description**
   - Mention every parameter in your description.
   - Describe the return value.
5. **Body**
   - Write the body of your function.
6. **Test**
   - Run the examples.

Step-by-step example

The problem:

The United States measures temperature in Fahrenheit and Canada measures it in Celsius. When travelling between the two countries it helps to have a conversion function.

Write a function that converts from Fahrenheit to Celsius.

1. Write examples (inside docstring)

```
>>> convert_to_celsius(32)
0
>>> convert_to_celsius(212)
100
```

2. Define parameter and return types

```
(number) -> number
```

3. Write function header (name and parameters)

```
def convert_to_celsius(fahrenheit):
```

4. Write function description (in a docstring)

Return the number of Celsius degrees equivalent to *fahrenheit* degrees.

We have so far:

```
def convert_to_celsius(fahrenheit):
    """
    (number) -> number
    Return the number of Celsius degrees
    equivalent to fahrenheit degrees.

    >>> convert_to_celsius(32)
    0
    >>> convert_to_celsius(212)
    100
    """
```

5. Write the body of the function

```
return (fahrenheit - 32) * 5 / 9
```

6. Call function to test its correctness using examples in docstring

```
convert_to_celsius(32)
convert_to_celsius(212)
```

Use case

English				metric		
1 **inch**				=	2.54	cm
1 **foot**	=	12	in.			
1 **yard**	=	3	ft.			
1 **rod**	=	5(1/2)	yd.			
1 **furlong**	=	40	rd.			
1 **mile**	=	8	fl.			

The United States uses the *English* system of (length) measurements. The rest of the world uses the *metric* system.

So, people who travel abroad and companies that trade with foreign partners often need to convert English measurements to metric ones and vice versa.

Develop the functions
inches->cm
feet->inches
yards->feet

Then develop the functions
feet->cm
yards->cm

Hint: Reuse functions as much as possible.