There different kinds of helpful „tools“ you can use during the design of a computer program. On this page I want to explain some of them.

**Types of tools**

* ***Debugging tools***

Isn’t going to end up in our final program, but helps us to develop our program faster and to find bugs in our algorithms

* + countcalls 🡪 Count calls to a function

def countcalls(f):

"""Decorator that makes the function count calls to it, in callcounts[f]."""

def \_f(\*args):

callcounts[\_f] += 1

return f(\*args)

callcounts[\_f] = 0

return \_f

callcounts = {}

@countcalls # fib = countcalls(fib)

def fib(n): return 1 if n <= 1 else fib(n-1) + fib(n-2)

* + trace 🡪 Prints the execution of a program

def trace(f):

indent = ' '

def \_f(\*args):

signature = '%s(%s)' % (f.\_\_name\_\_, ', '.join(map(repr, args)))

print('%s--> %s' % (trace.level\*indent, signature))

trace.level += 1

try:

result = f(\*args)

print('%s<-- %s == %s' % ((trace.level-1)\*indent, signature, result))

finally:

trace.level -= 1

return result

trace.level = 0

return \_f

@trace # fib = trace(fib)

def fib(n): return 1 if n <= 1 else fib(n-1) + fib(n-2)

* + disabled

Is another name for the identity function. This function returns its argument without doing any computation on it. This allows us to disable any decorator we have used in our program so far.

# ... all the different decorator definitions, like “def trace(...)”

def disabled(f): return f

trace = disabled

# ... all the function which use our decorators

* ***Performance*** ***tools***

Make your programs and algorithms faster

* + memoization (memo)

🡪 Caches return values of a function for future calls with the same parameters

def memo(f):

"""Decorator that caches the return value for each call to f(args).

Then when called again with same args, we can just look it up."""

cache = {} # dictionary

def \_f(\*args):

# try instead of if-else, because we would have to handle TypeError anyhow

try:

return cache[args]

except KeyError: # key not in dictionary

cache[args] = result = f(\*args)

return result

except TypeError:

# some element of args can't be a dict key

# e.g. a list can't be hashed in python, because its elements are mutable

return f(args)

return \_f

@memo # fib = memo(fib)

def fib(n): return 1 if n <= 1 else fib(n-1) + fib(n-2)

* ***Expressiveness tool***

Gives us more power to say more about our language

* + n\_ary

Turns a function with multiple parameters into a function with 2 parameters:

f(x, y, z) = f(x, f(y,z))

def n\_ary(f):

"""Given binary function f(x, y), return an n\_ary function such

that f(x, y, z) = f(x, f(y,z)), etc. Also allow f(x) = x."""

def n\_ary\_f(x, \*args):

return x if not args else f(x, n\_ary\_f(\*args))

return n\_ary\_f

@n\_ary # DECORATOR --> the same as seq = n\_ary(seq)

def seq(x, y): return ('req', x, y)

As you can see is it possible to implement all those “tools” in a way that

1. we do not have to change the functions which implement the actual functionality
2. the caller/user of a certain function doesn’t even need to know that we apply these tools

This is possible by using a design pattern called “[decorator](https://en.wikipedia.org/wiki/Decorator_pattern)”. The python language supports us implementing this pattern by offering the @ symbol. We can simple put it above the definition of a function. In this way we can save us the line:

function = decorator(function)

There is just one problem by using decorators: When we want to display some help in the python console it doesn’t shows us the help strings and function prototype of the function we want help for, but for the function the decorator returns. We take a look at the countcalls decorator example again to show the problem:

def countcalls(f):

"""Decorator that makes the function count calls to it, in callcounts[f]."""

def \_f(\*args):

callcounts[\_f] += 1

return f(\*args)

callcounts[\_f] = 0

return \_f

callcounts = {}

@countcalls # fib = countcalls(fib)

def fib(n):

“”” Returns the fibonacy sequence “””

return 1 if n <= 1 else fib(n-1) + fib(n-2)

If we now in the console enter the command: help(fib) we will get the following result:

In[1]: **help(fib)**

Help on function \_f in module \_\_main\_\_:

\_f(\*args)

This is definitely not want we want to see. Therefore we use another little decorator to overwrite the help string:

from functools import update\_wrapper

def decorator(d):

"Make function d a decorator: d wraps a function fn."

def \_d(fn):

return update\_wrapper(d(fn), fn)

update\_wrapper(\_d, d)

return \_d

@decorator

def countcalls(f):

...

The update\_wrapper(wrapper, wrapped) function updates the wrapper function so that it looks like the wrapped function.

And here is what we get now if we call help on the fib function:

In[2]: **help(fib)**

Help on function \_f in module \_\_main\_\_:

fib(n)

Returns the fibonacy sequence

**Language**

One of the most important tools is language. With this tools we are able to define a language we want to deal with instead of using the language that Python gives us.

* Grammar: Describes a language
* Interpreter: Interprets a language
* Compilers: Pre-Interpret the language, so that it can be read faster

***Sources:*** [Udacity course CS212](https://de.udacity.com/course/design-of-computer-programs--cs212/)