universität innsbruck

198.801

Introduction to Programming:

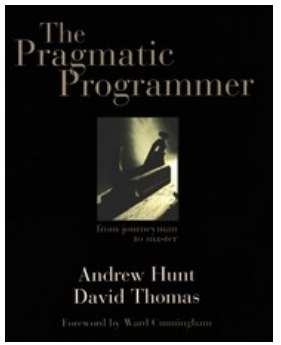
Programming in Python

Defining and Documenting Functions

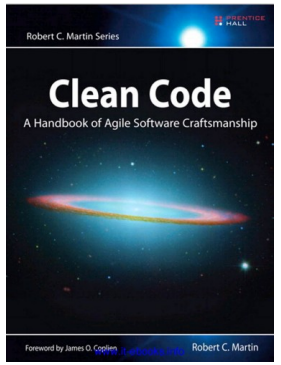
Joanna Chimiak-Opoka, PhD
University of Innsbruck, Digital Science Center

Goal

Learn about the
pragmatic programming principles:

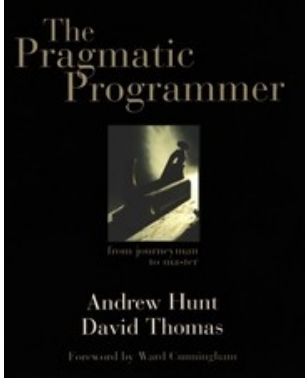


- DRY = don't repeat yourself
- characteristics of well-defined **functions**
- characteristics of well-written **comments**



Practice defining and documenting functions

The Evils of Duplications



Every piece of knowledge must have a

single,

unambiguous,

authoritative

representation within a system

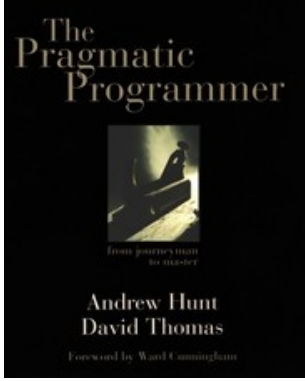
...*otherwise you have to remember to*

update all representations!

DRY principle → ***Don‘t Repeat Yourself***

[not *DRY* → *WET* → *Write Everything Twice*]

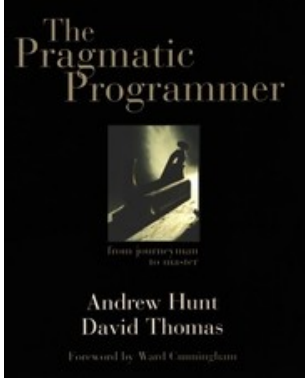
Origins of duplications

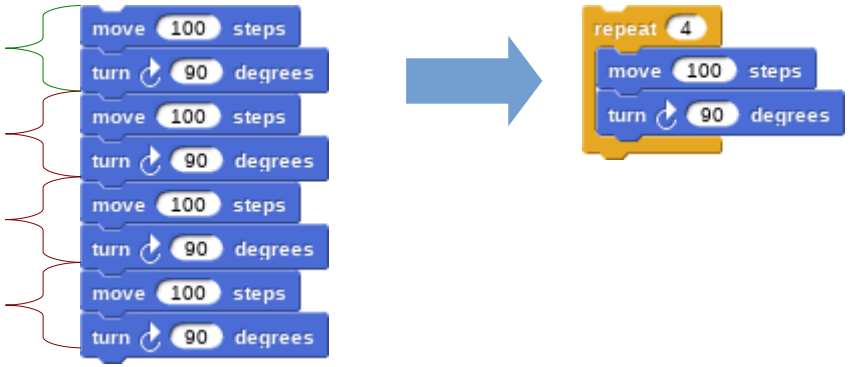


- **Imposed duplication**
  - developers **feel** they have **no choice**
  - the environment seems to require duplication.
- **Inadvertent duplication**
  - developers **don't realize** that they are duplicating information.
- **Impatient duplication**
  - developers get **lazy** and duplicate because it seems **easier**.

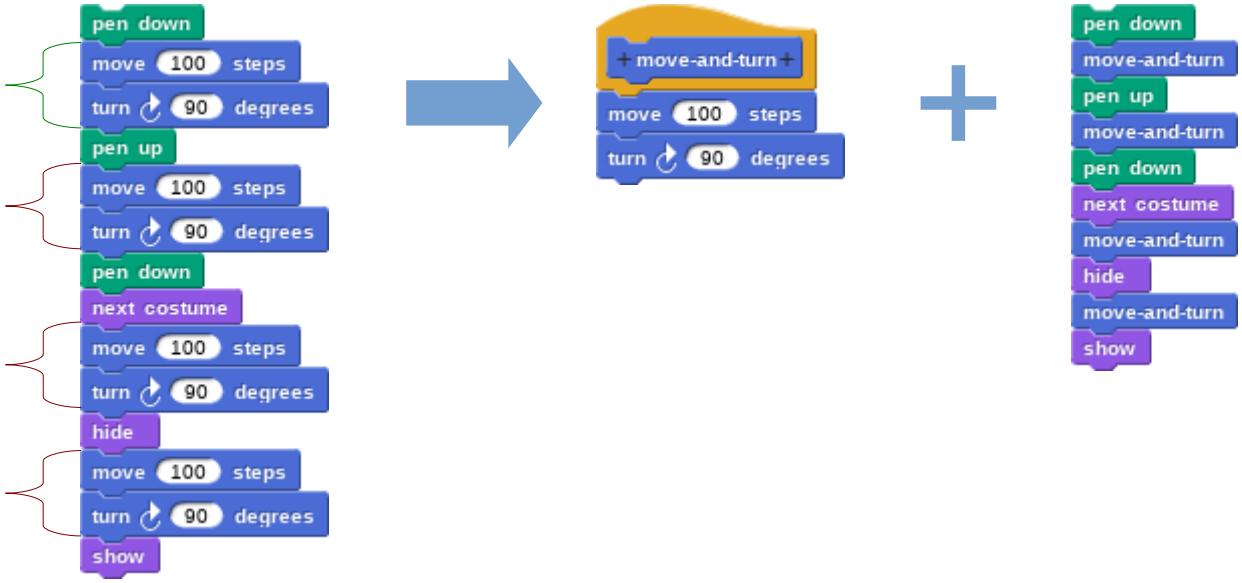
- **Interdeveloper duplication**
  - **multiple people** on a team (or on **different teams**) duplicate a piece of information.

Types of duplications in code
copy – paste



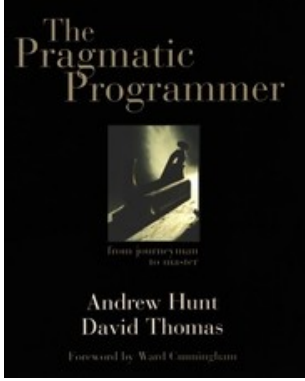


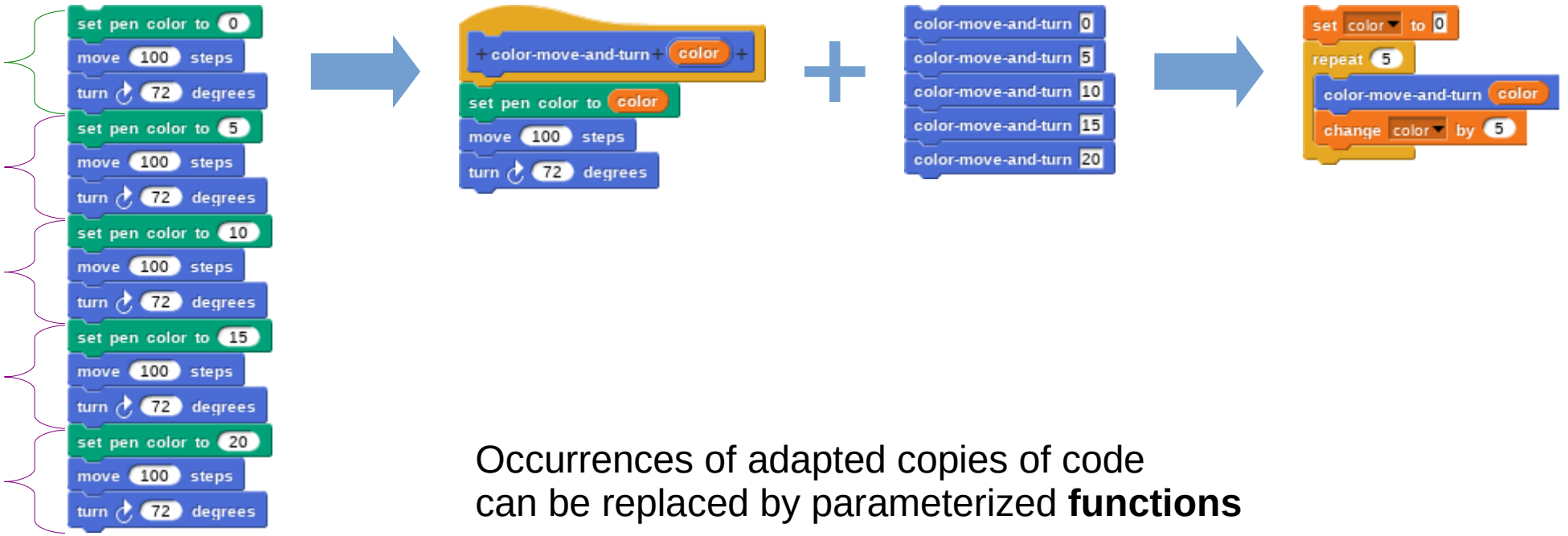
Subsequent occurrences of copied code can be replaced by loops (**iterations**)



Spread occurrences of copied code can be replaced by **functions**

Types of duplications in code
copy – paste – adapt

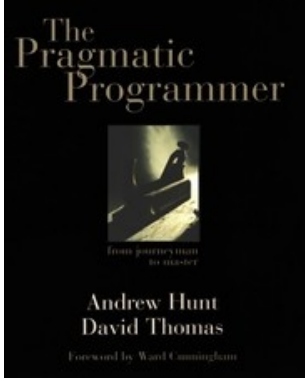


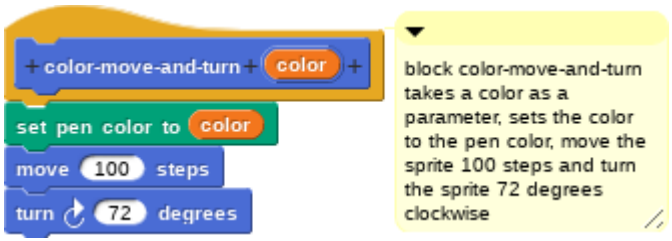


Occurrences of adapted copies of code
can be replaced by parameterized **functions**

The process of code improvement without changing its functionality
is called **refactoring** and should be supported by **tests**

Types of duplications across the **representations**

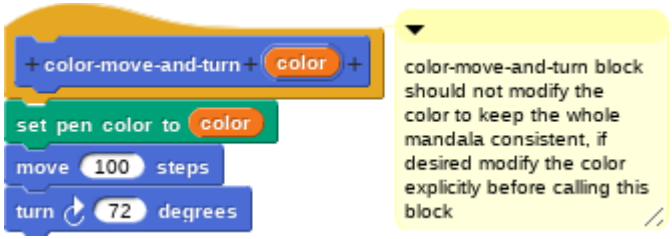




The script and the comment
consists **the same** piece of information

functionality = low level knowledge

WET

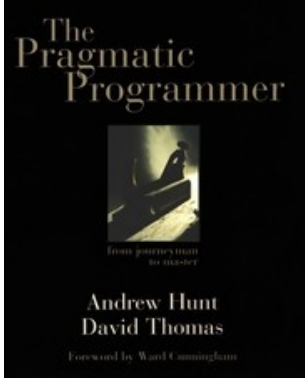


In the **script**
functionality (*low level knowledge*) is provided

in the **comment**
design *motivation* (*high level explanations*) is provided

DRY

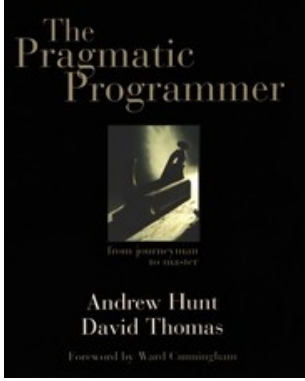
Avoiding Imposed Duplications



- Documentation in code
  - keep the **low-level knowledge** in the code
  - keep the **high-level explanations** in comments
    (see characteristics of good comments on the last slide)
  - **generation** of documentation from code

- Multiple representations of information
  - write a simple **filter** or a code **generator**
- Language issues e.g. headers and implementations
  - **header files** to document interface issues
  - **implementation files** to document the details that users of your code don't need to know

Avoiding Other Duplications



- Inadvertent Duplication
  - usage of **normalized** data

PERSONAL DATA 1
Name: ***Joanna***
Familiname: ***Chimiak-Opoka***
Address: ***Technikerstrasse 21a***

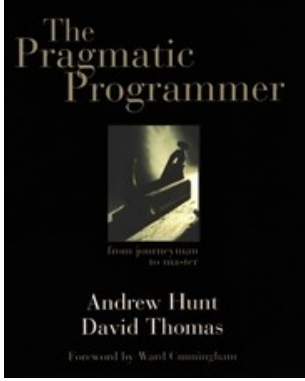
PERSONAL DATA 2
Name: ***Joanna***
Familiname: ***Chimiak-Opoka***
Address: ***Innrain 15***

PERSON 1
Name: ***Joanna***
Familiname: ***Chimiak-Opoka***

ADDRESS 1
Postal address: ***Technikerstrasse 21a***

ADDRESS 2
Postal address: ***Innrain 15***

Avoiding Other Duplications



- Inadvertent Duplication
  - usage of **normalized** data
  - usage of method instead of **derived values** (performance!)

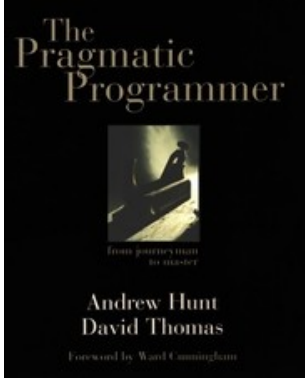
weight = 60
height = 170

bmi = 20.8

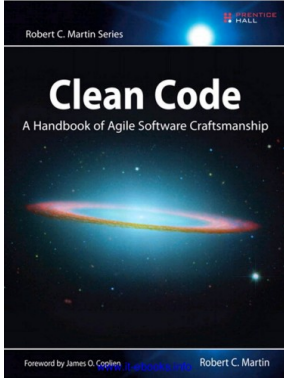
weight = 60
height = 170

bmi(weight, height)

Avoiding Other Duplications



- Inadvertent Duplication
  - usage of **normalized** data
  - usage of method instead of **derived values** (performance!)
- Impatient Duplication
  - **spend time up front** to save pain later
  - use **parametrization** to avoid copy–past–adapt
- Interdeveloper Duplication
  - a **clear design** with a well-understood division of **responsibilities** within it
  - **communication** (information flow, history)
  - **shared utility routines**



Characteristics of functions

- **Functionality**
  - **do one thing** - do it well, but only this
    - **command / query separation**

      either do something or answer something, but not both.
    - without **side effects**
  - **small**, the smaller the better
  - one level of **abstraction** per function

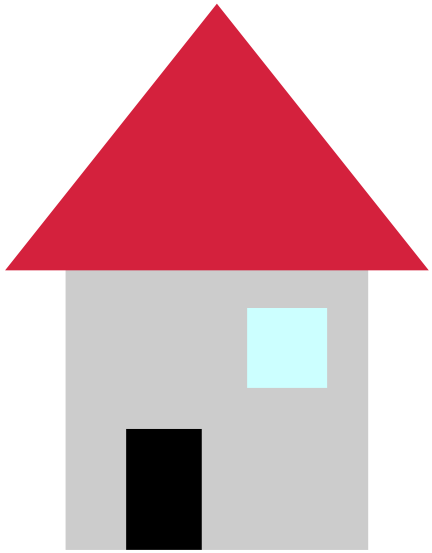
    check: the stepdown rule:
    reading code from top to bottom like a narrative
- **Signature**
  - with descriptive **names**
  - with low number of **parameters**, preferably 0-2

Stepdown Rule: Example

- The same level of abstraction
  - domain-independent
    - draw a rectangle
    - draw a triangle
    - draw a rectangle
    - draw a square
  - domain-specific
    - draw a wall
    - draw a roof
    - draw a door
    - draw a window

good readability

even better readability

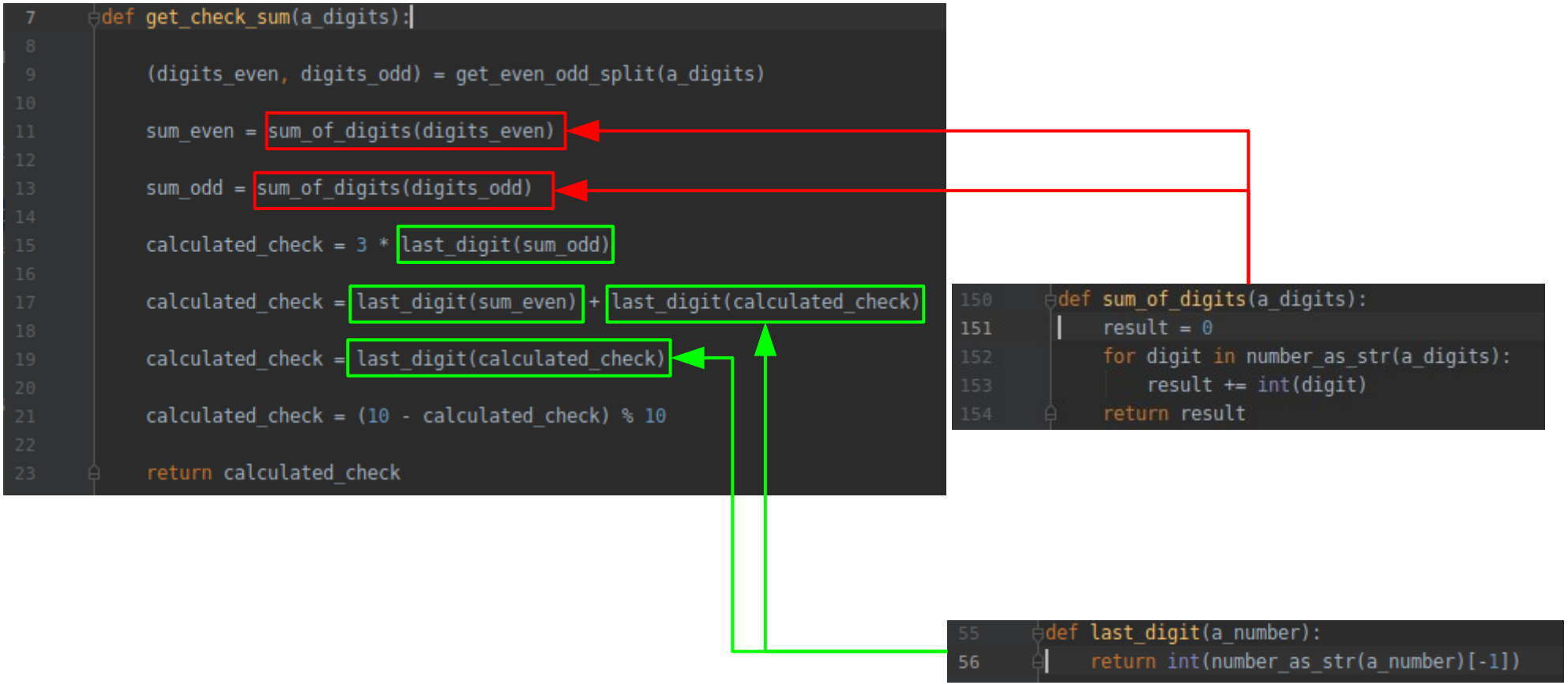


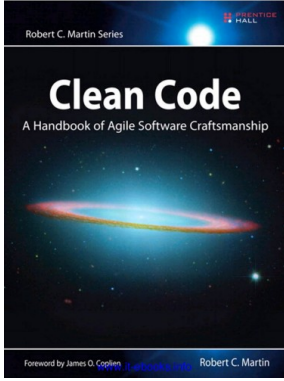
draw a hause

- Mixed levels of abstraction
  - draw a rectangle
  - repeat 3 times
    - draw a line
    - turn 60 degrees
  - draw a door
  - repeat 4 times
    - draw a line
    - turn 90 degrees

bad readability

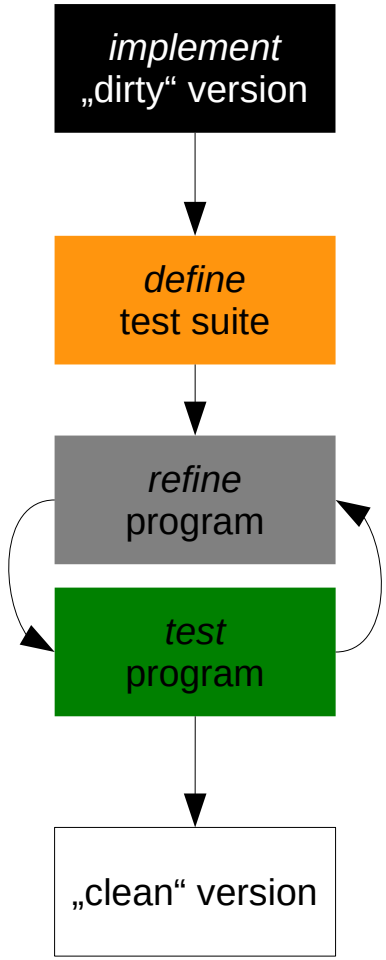
Stepdown Rule: Python Example





How to write functions

- **Implement** initial, “dirty” version:
  - long and complicated,
  - with lots of indenting and nested loops,
  - with long argument lists and arbitrary names,
  - with duplicated code…
- Write a suite of (**unit**) **tests**
  - that cover whole functionality
- **Refine** the code… while keeping the tests passing
  - splitting out functions,
  - changing names,
  - eliminating duplication,
  - shrink the methods and reorder them….
- **Final “clean” version** of functions short, well named, and nicely organized and following other rules



Scope of variables

Scope refers to the **visibility** of variables.
In other words, which parts of your program can see or use it.

- **Built-in scope** with predefined names

  Examples of function and variables names: open(), len(), __name__.

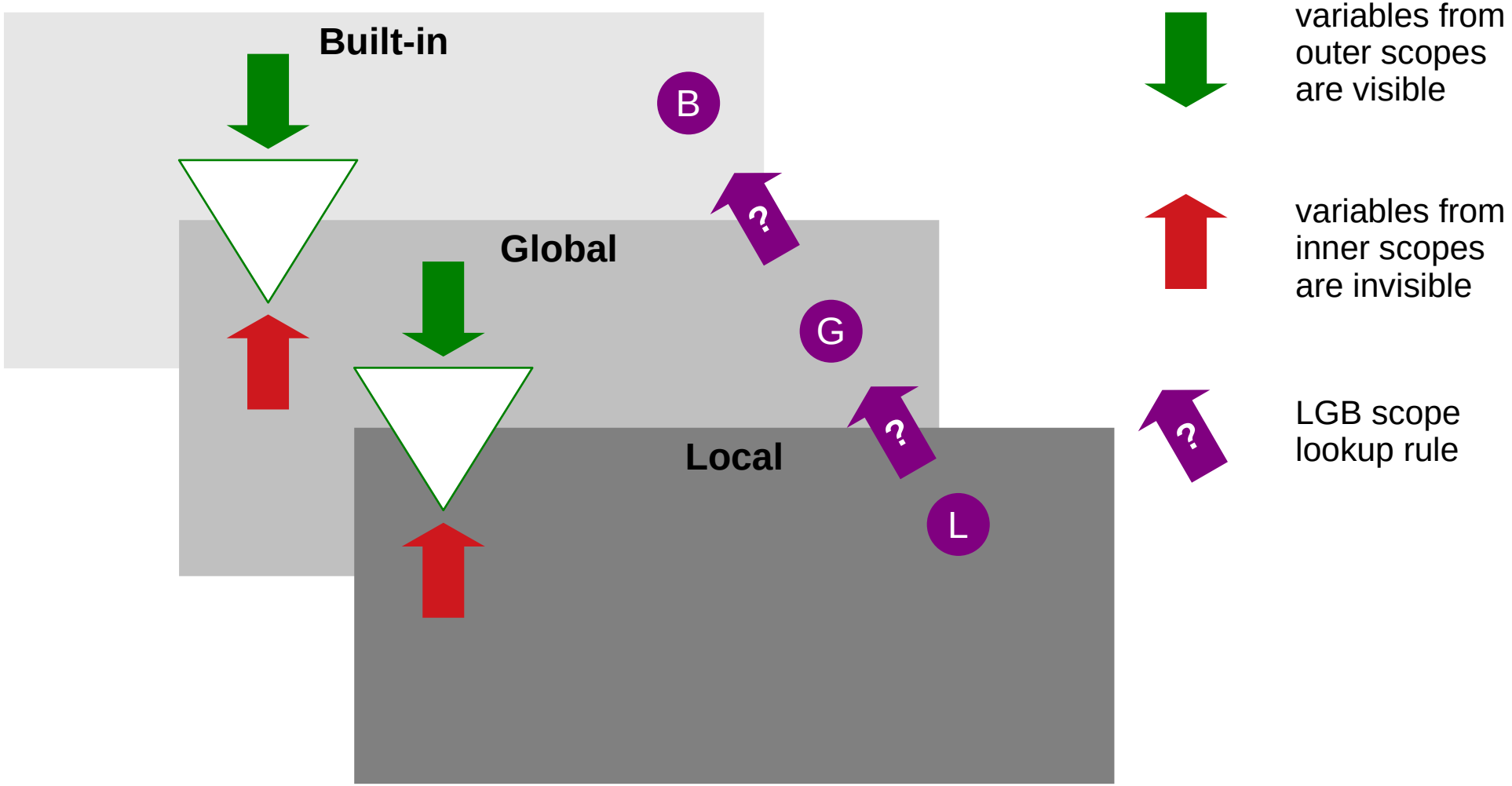
- **Global scope** with names defined at top-level of a module.

  Once defined, every part of your program can access a variable.

- **Local scope** with names defined withnin a function.

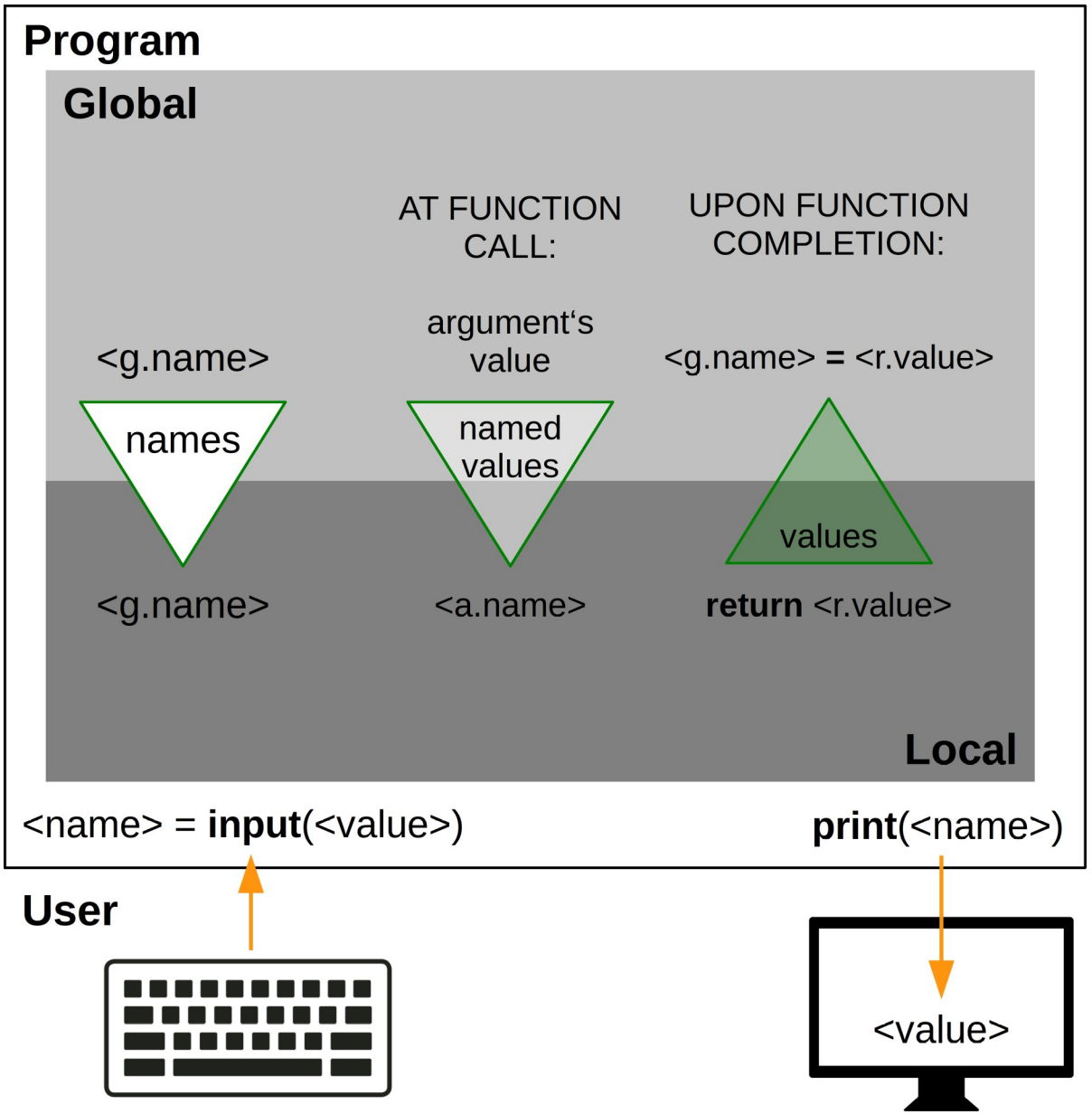
  If defined in a function variable‘s scope is limited to this single function.

Relations between scopes

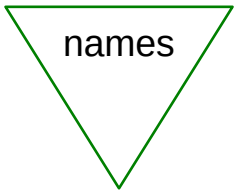


REMARK: when an **exception** is raised it, it goes to the outer scopes until it is caught or shown to a user

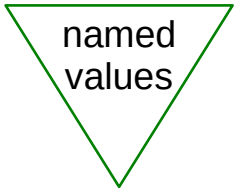
Exchanging information



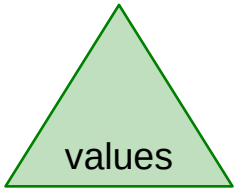
INTERNAL COMMUNICATION



variables from
outer scopes
are visible



values from
an outer scope
can be passed
as arguments



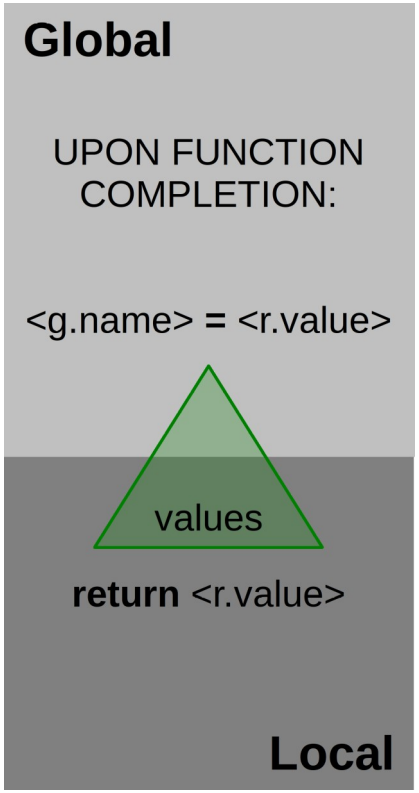
values from
a local scope
can be passed
to its enclosing scope

COMMUNICATION WITH A USER

using **input**() and **print**()

program cannot
access printed values

Returning a value



Returning a value from a function and **using** it in a program

```
def fun():
    a = 3
    return a
```

… in an **assignment**

```
b = fun()
```

b: 3

... in a **function call**
warning: it may be lost for a program

```
print(fun())
```

3

not using it at all / ignoring it
it gets **lost** for the program

```
fun()
```

Printing a value from a function, the value is **lost** for a program

```
def fun():
    a = 3
    print(a)
```

3

```
b = fun()
```

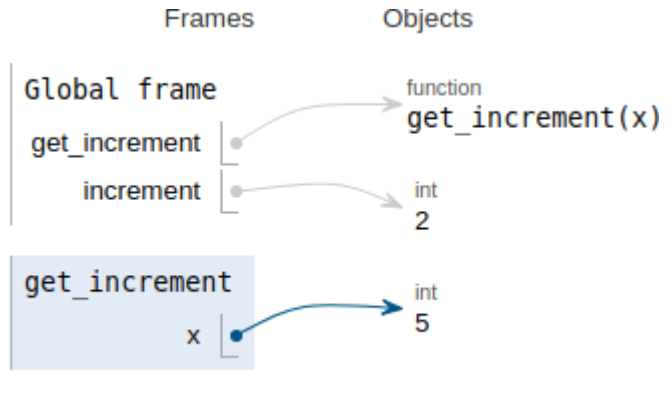
b: None

Scopes: Examples

- local and global scopes
- name shadowing
- declaring names global

Python 3.6

```
1  def get_increment(x):
2      return x+increment
3
4  increment = 2
5
6  x = get_increment(5)
7
8  x = get_increment(7)
```

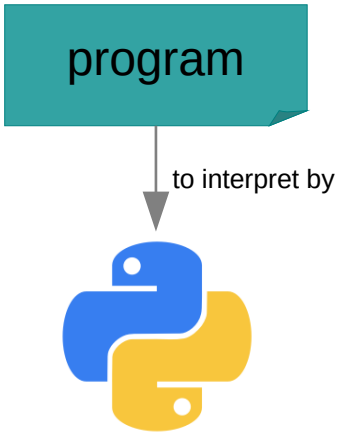


Program Interpretation

Direct interpretation

of a program

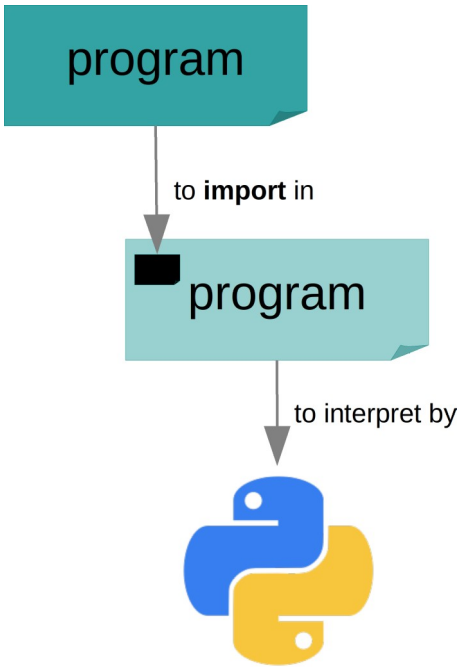
as so called **main** program



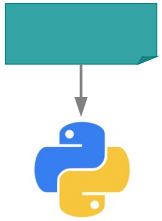
Indirect interpretation

of a program

as so called **module**



Usage Scenarios



All-inclusive solution,

everything is implemented

in one file (program)

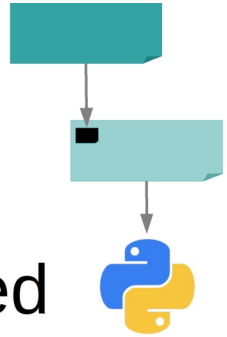
- for a small problem

  possible and acceptable

- for a large problem

  in bad style or

  even infeasible



Modular design,

functionality spread

over **a number of files**

- recommended

  for a large problem

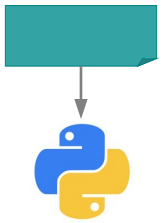
- importing third-party or

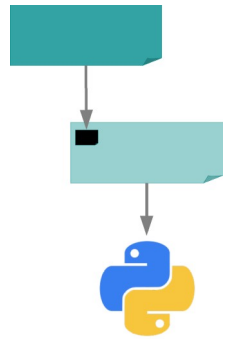
  own module / package

*A **module** is a single file (or files) that can be imported.*

*A **package** is a collection of modules in directories that give a package hierarchy.*

Technical Solution





function and
variable definitions

task.py

```
# LIBRARY

def sqfeet_to_sqmeters(a_value):
    pass


def sqmeters_to_sqfeet(a_value):
    pass
```

run this program
python3 task.py

use the definitions
import task

`__name__ == '__main__'` *is **True***

```
if __name__ == '__main__':
```

`__name__ == '__main__'` *is **False***

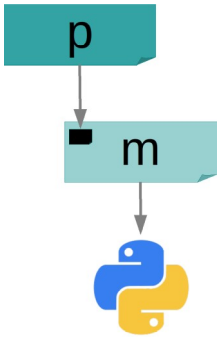
the True-branch will be executed

```
# MAIN
# the top-level
# script environment
pass
```

the True-branch will be not executed

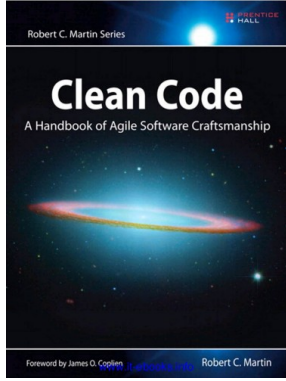
user interaction
testing

More Technical Remarks



- **import** p
  - basic import command
  - requires name space
  - here: p.my_function()
  - in general: <*package name*>.<*function name*>()
- **from** p **import** *
  - acceptable only for small, own packages
  - visible inside *m* name space
  - my_function()
- **import** matplotlib.pyplot **as** plt
  - handy for modules with long names or within packages
  - here: plt instead of matplotlib.pyplot
  - here: plt.plot()
- **from** matplotlib.pyplot **import** plot
  - recommended for large modules to avoid
    - unexpected name clashes (as with import *) and
    - loading large amount of unused code (as with import)
  - here: plot()

Characteristics of comments



- **Spare comments**
  - the code should be so clear and expressive that it does not need the comments at all
  - **explain yourself in code** through descriptive names and clean structures

Good comments

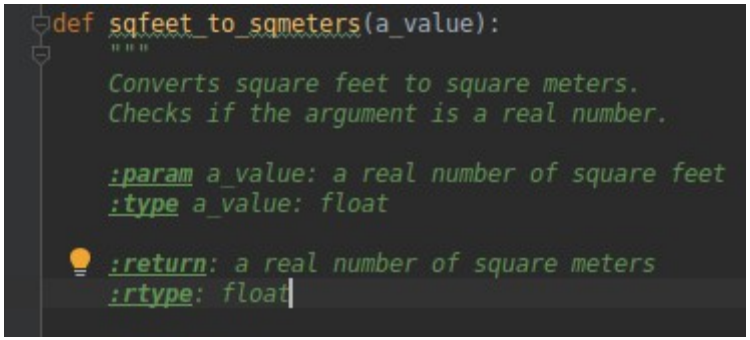
- **Legal**: copyrights, licence, …
- **Clarification**:
  - improve readability of code (e.g. re)
  - **intent** behind a decision (why?)
  - details about attempt
- **Communication**:
  - **warnings** to other programmers,
  - **to do** comments,
  - **amplification** of importance,
  - **documentation** of application public interfaces (APIs)

Bad comments

- **Content**
  - **unclear** meaning, forces you to look in another module for the meaning of it
  - **redundant**, the same information can be read from code
  - **misleading**, makes harder to read and understand the code
- **Format**
  - **commented-out code**
  - **formatted** comments, e.g. HTML
- …

Documentations Strings

- Documentation in code
  - keep the **low-level knowledge** in the code
  - keep the **high-level explanations** in comments
  - **generation** of documentation from code
- **Documentation** of application public interfaces (**API**s)



docstrings
– Python documentation strings

provide a convenient way of associating documentation with Python modules, functions, classes, and methods.
An object's docsting is defined by including a string constant as the **first statement** in the object's definition.

docstrings

```
def sqfeet_to_sqmeters(a_value):
    """
    Converts square feet to square meters.
    Checks if the argument is a real number.

    :param a_value: a real number of square feet
    :type a_value: float

    :return: a real number of square meters
    :rtype: float

    Examples:
    Correct input type:
    >>> sqfeet_to_sqmeters(107.64)
    10.0
    >>> sqfeet_to_sqmeters(10.764)
    1.0
```

```
>>> sqfeet_to_sqmeters('txt')
Traceback (most recent call last):
Expected type 'float', got 'str' instead more... (Ctrl+F1)
```

Used in static analysis to generate hints

Quick Documentation

```
def sqfeet_to_sqmeters(a_value):
    """
    Conver
    Checks

    :param
    :type

    :retur
    :rtype

    Exampl
    Correc
    >>> sq
    10.0
    >>> sq
    1.0
    >>> sq
```

```
Programs.SqFeet.task
def sqfeet_to_sqmeters(a_value: Any) -> Any

Converts square feet to square meters.
Checks if the argument is a real number.

Examples:
Correct input type:
>>> sqfeet_to_sqmeters(107.64)
10.0
>>> sqfeet_to_sqmeters(10.764)
1.0
>>> sqfeet_to_sqmeters(0)
0.0
```

Auto-Generated HTML Documentation

Module task

source code

This module provides two functions: sqfeet_to_sqmeters() and sqmeters_to_sqfeet().

Examples: >>> sqfeet_to_sqmeters(sqmeters_to_sqfeet(100)) 100.0 >>> round(sqfeet_to_sqmeters(sqmeters_to_sqfeet(1000)), 10) 1000.0

Functions	
	sanitize_parameter(a_value) TODO: write docstring for this function TODO: write doctest for this function
	sqfeet_to_sqmeters(a_value) Converts square feet to square meters.
	sqmeters_to_sqfeet(a_value) TODO: write docstring for this function TODO: write doctest for this function

Variables	
	ERROR_NEGATIVE = 'The argument must be a non-negative number'
	ERROR_NAN = 'The argument must be a number'
	__package__ = None

Function Details

sqfeet_to_sqmeters(a_value)

Converts square feet to square meters. Checks if the argument is a real number.

:param a_value: a real number of square feet :type a_value: float

:return: a real number of square meters :rtype: float

Examples: Correct input type: >>> sqfeet_to_sqmeters(107.64) 10.0 >>> sqfeet_to_sqmeters(10.764) 1.0 >>> sqfeet_to_sqmeters(0) 0.0

Incorrect input range or type: >>> sqfeet_to_sqmeters(-1) Traceback (most recent call last): ... ValueError: The argument must be a non-nega argument must be a number >>> sqfeet_to_sqmeters('10') Traceback (most recent call last): ... TypeError: The argument must be a number >> must be a number

Raise an Exception

- **Python interpreter** **raises an exception** when there is an Error in a program and a user of the program see the **traceback** information

- **As a programmer**, you can raise an exception of a given **type** and with a given **message**, for example

```
raise ValueError("a_value must be a positive number")
```

- Customized message are related to your program in opposite to standard messages which are related to Python language. As such they are more helpful for the user.