Functions, Recursion, and Files

Dr. Sebastian Proksch

University of Zurich, Institute of Informatics

Outline

- Functions
    - Declaration
    - Calling Functions
- Recursion
- Working With Files
- Exercises

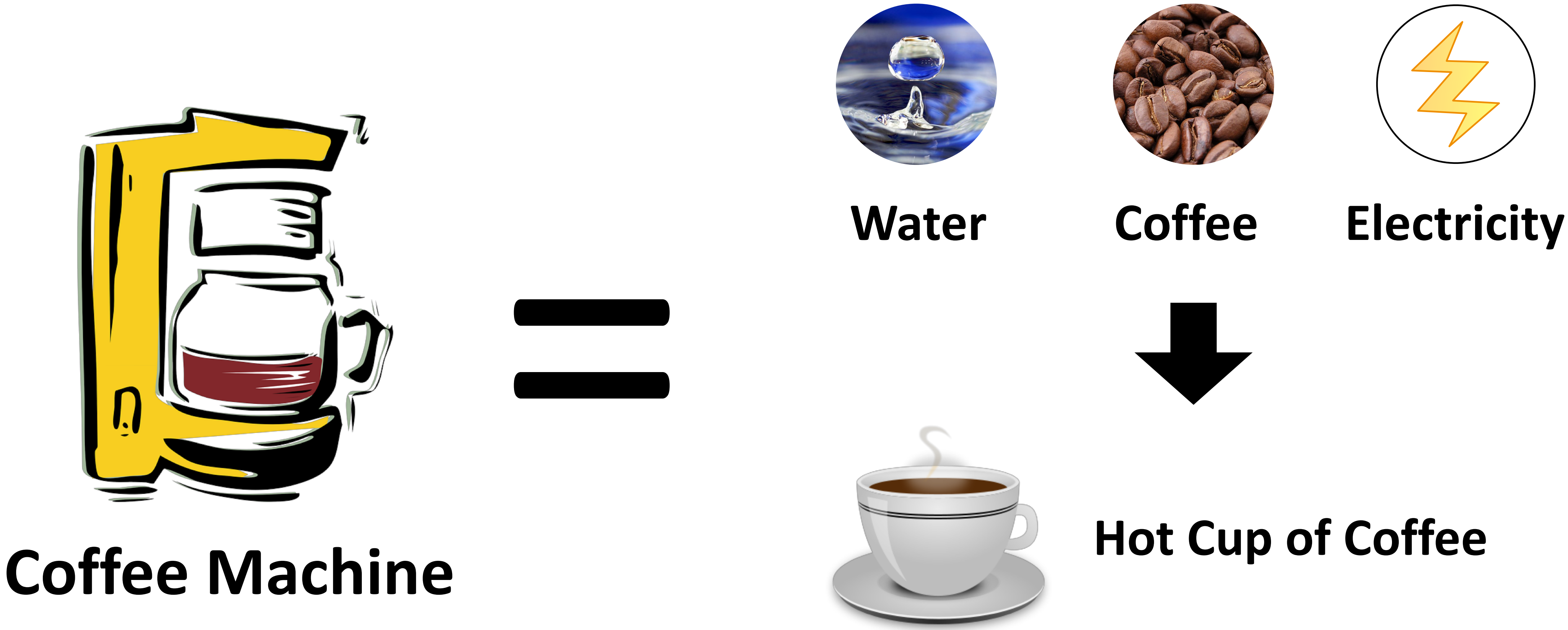
Motivation

- So far we have written code in a single file and executed it
- This approach does not scale to real software systems which can contain hundreds of thousands and even millions of lines of code
- We need a better way to structure and organize the code
- For this we use mechanisms of abstraction and decomposition

Abstraction

- Knowing how to use something without knowing how it actually works (how it is implemented)
- Usage defined by its *interface*
  - Requirements/Assumptions on input
  - Guarantees for output
- Functionality is provided in a black box

Abstract Description of a Coffee Machine



Coffee Machine

Water

Coffee

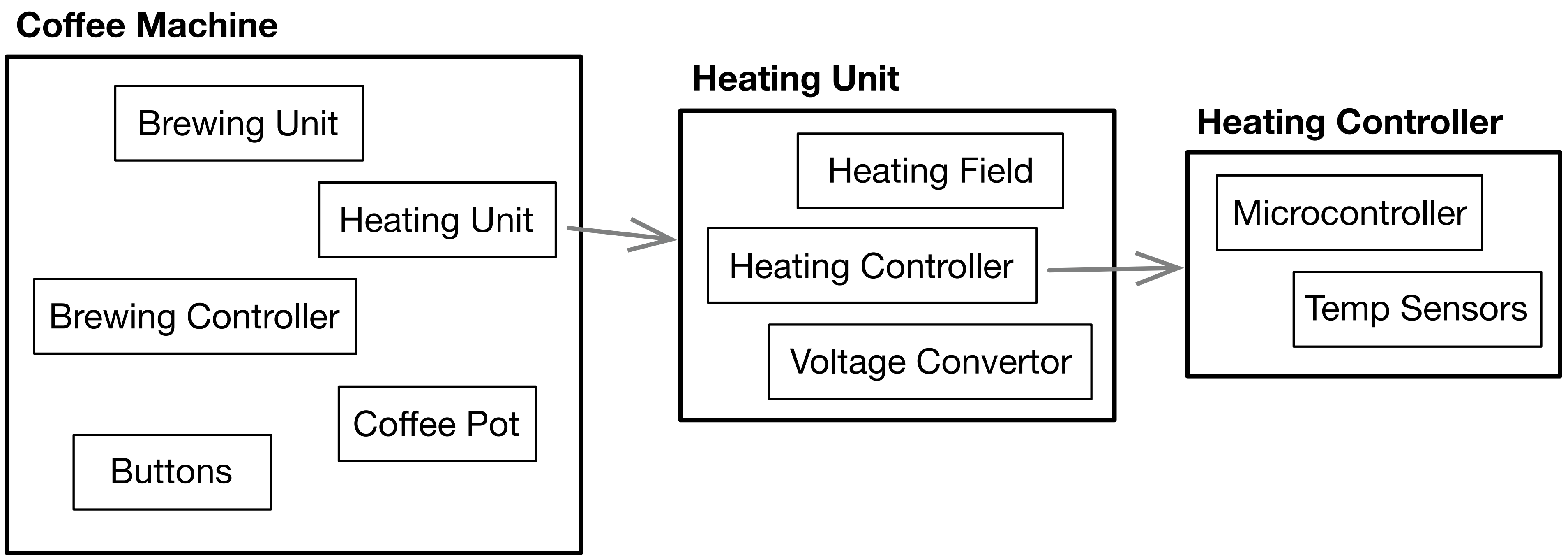
Electricity

Hot Cup of Coffee

https://en.wikipedia.org/wiki/File:Water_drop_001.jpg, https://en.wikipedia.org/wiki/File:Roasted_coffee_beans.jpg, https://commons.wikimedia.org/wiki/File:Coffee_cup_icon.svg

Decomposition

- Break a problem in different, self-contained parts that can be implemented and **re-used** separately



Functions

- First mechanism for abstraction and decomposition
- You have already used functions without knowing how they are implemented

```
>>> name = input('What is your name?')
>>> age = int(“123”)
>>> size = len(name)
>>> import time
>>> time.time()
>>> range(1, 10)
```

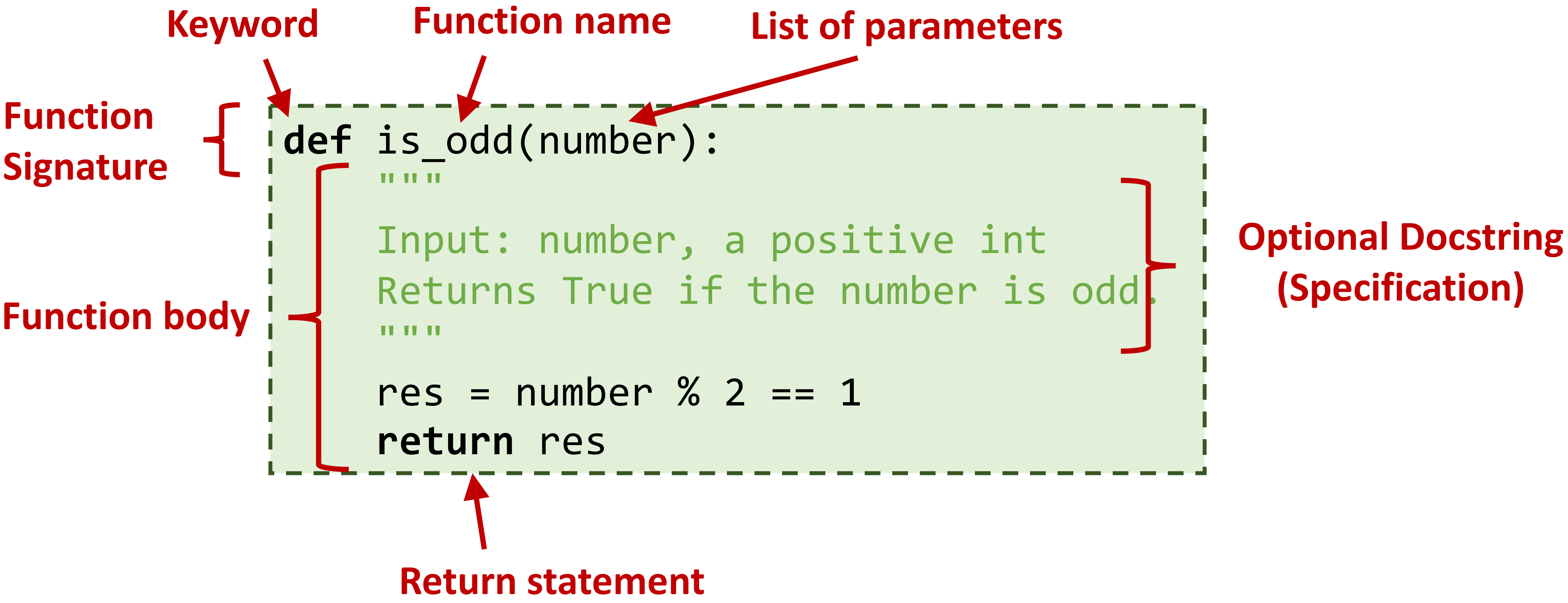
Function Definition

Function Definition

- A function is a **named** set of instructions that performs a specific task.
- A function can take one or more **parameters** (arguments) and **returns** a value.
- You can have as many instructions as you want in a function
- In Python, the nested instructions have to be indented (“function body”)
- Functions are not executed until they are **called** (**invoked**) in the program.

```
def «NAME» («LIST OF PARAMETERS»):
   «INSTRUCTIONS»
```

Function Definition Example



Keyword

Function name

List of parameters

Function Signature

```python
def is_odd(number):
    """
    Input: number, a positive int
    Returns True if the number is odd.
    """
    res = number % 2 == 1
    return res
```

Function body

Optional Docstring (Specification)

Return statement

Calling Functions

- Once defined, functions can be called
- Caveat: You cannot use a function before its definition

```
def sum(a, b):
    return a + b

s = sum(1, 2)
print("sum = %d" % s)
```

Advantages of Defining Functions

- Functions enable decomposition and abstraction!
  - Facilitates code reuse
  - Helps avoiding repetitive code
  - Hides implementation details
- Makes testing easier

Trick: Use the Pass Statement

- pass is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.

```
def empty_function():
    pass

while True:
    pass
```

Trick: Understanding a Function

```
>> help(round)

round(number[, ndigits]) -> number

Round a number to a given precision in
decimal digits (default 0 digits). This
returns an int when called with one
argument, otherwise the same type as the
number. ndigits may be negative.
```

Return Statement

Return values

- functions can return a value to the caller using the **return** instruction

```
def compute_sum(a, b):
    return a + b

s = compute_sum(10, 10)
print("computed sum: %d" % s)
```

Return values

- you can have multiple return instructions inside a function
- Only one return will be executed

```
def absolute_value(a):
    if a < 0:
        return -a
    else:
        return a

print(absolute_value(-3))  # prints 3
print(absolute_value(4))   # prints 4
```

Return values

- if you do not return anything, None will be returned by default

```
def absolute_value(a):
    if a < 0:
        return -a
    elif a > 0:
        return a

print(absolute_value(0)) # prints None
```

Function Parameters

- Function definitions **can** include one or more **formal parameters**
- When a function is called you must pass **actual parameters** to all declared parameters (... that do not have a default value)

```
# msg is a formal parameter
def print_msg(msg):
    print(msg)

# "Hello, World!" is an actual parameter
print_msg("Hello, World!")
```

Calling Functions

You can pass literals (e.g. 1, 2, 3) or variables in a function call...

```python
def compute_sum(a, b, c):
    return a + b + c

a = 1
b = 2
c = 3
s = compute_sum(a, b, c)
print("sum = %d" % s)
```

Calling Functions

... or any kind of expression, which is then evaluated before the call.

```
def compute_sum(a, b):
    return a + b

s = compute_sum(3, 2 - 1)

print(s) # 4
```

Default Arguments

You can specify a default value for one or more parameters.

```
def greet(name=“Stranger”):
    print("Hello, %s!" % name)

greet("Alice")    # Hello, Alice!
greet()           # Hello, Stranger!
```

Keyword Arguments

- you can modify the order in which you pass the arguments by specifying the name

```
def greet(name, msg):
    print(msg % name)

greet("Alice”, “Hello, %s!”)                 # Hello, Alice!
greet(msg="Goodbye, %s!", name="Alice")    # Goodbye, Alice!
```

Functions Are First Class Objects

```
def fun():
   return 3.1415

i = 5
s = “foo”
f = fun

type(i) # int
type(s) # str
type(f) # function
```

Passing Functions as Arguments

```
import random
def rndGen():
   return random.randint(1, 100)

def zeroGen():
   return 0

def fun(numGen):
     print(“number: %d” % numGen())

fun(rndGen)  # number: «Random Number Between 1 and 100»
fun(zeroGen) # number: 0
fun(rndGen)  # number: «Random Number Between 1 and 100»
```

Anonymous Functions

```
lambda «LIST OF PARAMETERS»: «EXPRESSION»
```

```
double = lambda n: n*2

print(double(2)) # prints 4
```

Functions as First Class Objects

- can be passed as arguments to other functions
- can be returned as values from other functions
- can be assigned to variables
- can be stored in data structures (more about it later)

Variable Scoping

Variable Scope

- the scope is a mapping from names to values
- when you create a variable outside a function, we say the variable was defined inside the **global scope**

```
# global scope
x = 10
print("x = %d" % x) # x = 10
```

Variable Scope

- when you define a variable inside a function, it is available in the **function scope**

```
def f():
    # function scope
    x = 20
    print(x)            # (1) x?

# global scope
x = 10
print(x)                # (2) x?
f()
print(x)                # (3) x?
```

Variable Scope

- variables that are defined inside a function cannot be accessed from outside of the function

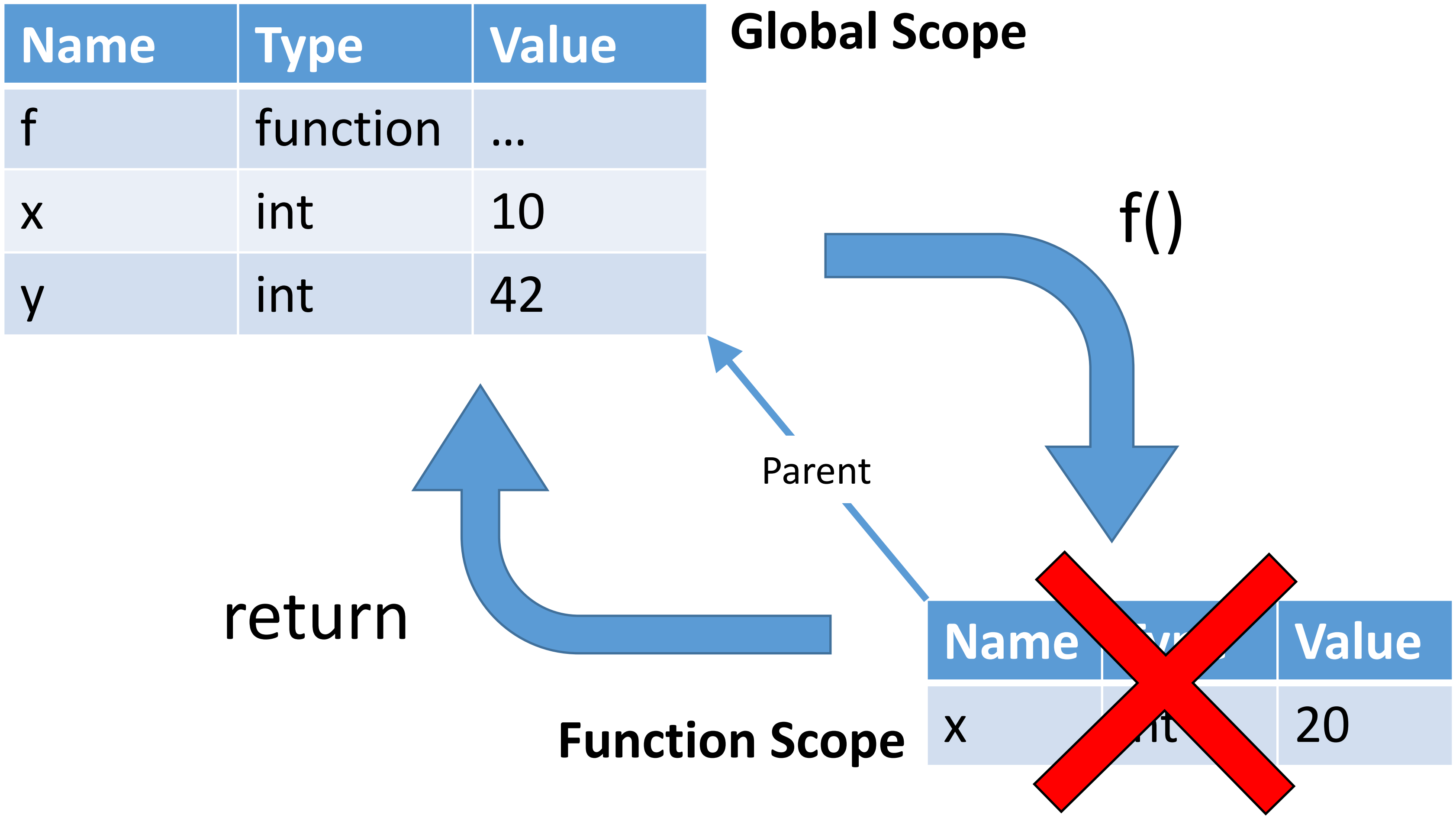
```
def f():
    x = 20
    print(x)    # 1: x is 20

f()
print(x)        # 2: NameError: name 'x' is not defined
```

Scoping

```
def f():
    x = 20
    print(x)
    print(y)

x = 10
y = 42
print(x)
f()
print(x)
```



Global Scope

Name	Type	Value
f	function	…
x	int	10
y	int	42

Function Scope

Name	Type	Value
x	int	20

Python Tutor

- to better understand how the different scopes (or frames) work, please use http://www.pythontutor.com/ to try different examples

Recursion

Recursion

- A function is recursive if it **calls itself**
- Break a problem into smaller versions of the same problem
- You must specify a **base case** (a small version of the problem that can be solve directly) to avoid infinite recursion
- Can be re-written iteratively

Functions Can Call Themselves!

```python
def r():
    r()
r()
```

```
Traceback (most recent call last):
File ”examples.py", line 3, in <module>
   r()
File "examples.py", line 2, in r
   r()
File "examples.py", line 2, in r
   r()
File "examples.py", line 2, in r
   r()
[Previous line repeated 995 more times]
RecursionError: maximum recursion depth exceeded
```

Computing a^b Recursively

- Recursion Anchor: $a^0 = 1$
- Recursion Step: $a^b = a * a^{b-1}$

```
def power(a, b):
    if b == 0:
        return 1

    return a * power(a, b-1)

power(2, 3)
```

Recursive Execution

Program Definition

```
def power(a, b):
  if b == 0:
    return 1
  return a * power(a, b-1)

power(2, 3)
```

Stepwise Execution

```
power(2, 3)
2 * power(2, 2)
2 * 2 * power(2, 1)
2 * 2 * 2 * power(2, 0)
2 * 2 * 2 * 1
2 * 2 * 2
2 * 4
8
```

Computing a^b Iteratively

```python
def power(a, b):
    result = 1
    for i in range(b):
        result *= a

    return result

print(power(2, 3)) # prints 8
print(power(2, 0)) # prints 1
print(power(2, 1)) # prints 2
```

Working With Files

Using Files

- so far we have either used `input` or hard-coded data in programs
- after executing program all generated results are lost
- sometimes we would like to work with more persistent data, i.e., read data from an existing file and/or save results to a file

Basic Operations on Files

- create a new file
- open a file
- read/write
- close a file ???

Operations on Files – Create/Open

```
# ‘test.txt’ is in the same directory as the python file

# open file for reading with 'r’
f1 = open('test.txt', 'r')

# opening a file in write mode will either create a new file
# (if the file does not exist) or overwrite an existing one
f2 = open('test_2.txt', 'w')
```

Operations on Files - Reading

```
f = open('test.txt', 'r')
# read the entire file content
contents = f.read()
print(contents)
```

```
f = open('test.txt', 'r')
# read only the first 10 characters of the file
contents = f.read(10)
print(contents)
```

Operations on Files – Reading by Line

```
f = open('test.txt', 'r')
# read first line of the file
line = f.readline()
print(line)
```

```
f = open('test.txt', 'r')
# read the contents of the file line by line
for line in f.readlines():
    print(line)
```

Operations on Files - Writing

```
# creating a new file for writing (or over-writing)
f = open('test.txt', 'w’)
f.write('First line’)
f.write(‘Still the First line\n’)
f.write('Second line\n')
```

```
# open file for appending with 'a’
f = open('test.txt', 'a’)
f.write(‘This string is added... ’)
f.write(‘to the existing content!')
```

Operations on Files - Closing

```
# open file for writing with 'w’
f = open('test.txt', 'w’)
f.write(‘...’)
# to save the changes to the file and make
# it available for reading, it must be closed!
f.close()
```

With Statement

The `with` statement will bind the opened file handle to a variable and it will automatically handle the closing at the end of the block.

```
with open('test.txt', 'w') as f:
    f.write(”...")

with open('test.txt', 'r') as f:
    print(f.read())
```

Take Home Message

After todays lecture, you should know...

- the value of abstraction and decomposition
- how to define functions, both named and anonymous
- how to call functions, pass parameters, use return statements
- the difference between formal and actual parameters
- what is a scope? Variable scope? Function Scope? Global Scope?
- the concept of recursive functions
- how to write/append text to files
- how to read text from files

Exercise 1

Sum-Up Digits

Exercise 1

Implement a function that returns the sum of the digits of an integer.

Examples:

1234 → 1 + 2 + 3 + 4 = 10

100001 → 1 + 1 = 2

Exercise 2

Fibonacci Numbers

Exercise 2

Compute the n-th number of the Fibonacci sequence.
(Hint: use a recursive function)

Definition:

f(0) = 1

f(1) = 1

f(n) = f(n-1) + f(n-2)

https://en.wikipedia.org/wiki/Fibonacci_number

Exercise 3

Implement a function that copies the contents of a file to a new file. The function should add a header to the top.

Signature: copy_file(src, dest, header)