Midterm-Exam Information

Dr. Carol Alexandru-Funakoshi

University of Zurich, Department of Informatics

Midterm Exam, what, when and where?

- Midterm exam: Monday, 07.11.2022, 18:15 - 19:15
  - Content: all material of the first 6 weeks (slides up to and including "Lists, Tuples & Dictionaries")
  - https://uzh.inspera.com - You **must** use your own programming environment!
  - You **must attend** the exam! If you do not attend the midterm exam, you are excluded from attending the final exam.
  - You **do not have to pass** the midterm exam. There is no pass/fail, you just earn points toward a bonus.

Bonus System

- Your grade is primarily determined by your **final exam**
- Students that pass the final exam can further improve their grade in a bonus system. For both the midterm and the assignments, apply the following rule:
  - If you reach at least 65% of the achievable points, every additional 1% gives you 0.01 bonus points up to a maximum of 0.25 (at 90% points achieved)
    - Example: Midterm 74% of points achieved, assignments 87% of points achieved, 5.00 in the final exam ➞ 0.09 + 0.22 = 0.31 bonus; Final grade: 5.25
    - There will be *roughly* 100-120 points achievable in the assignments
- The bonus **cannot** help you to pass the final exam though!

Exam – Important

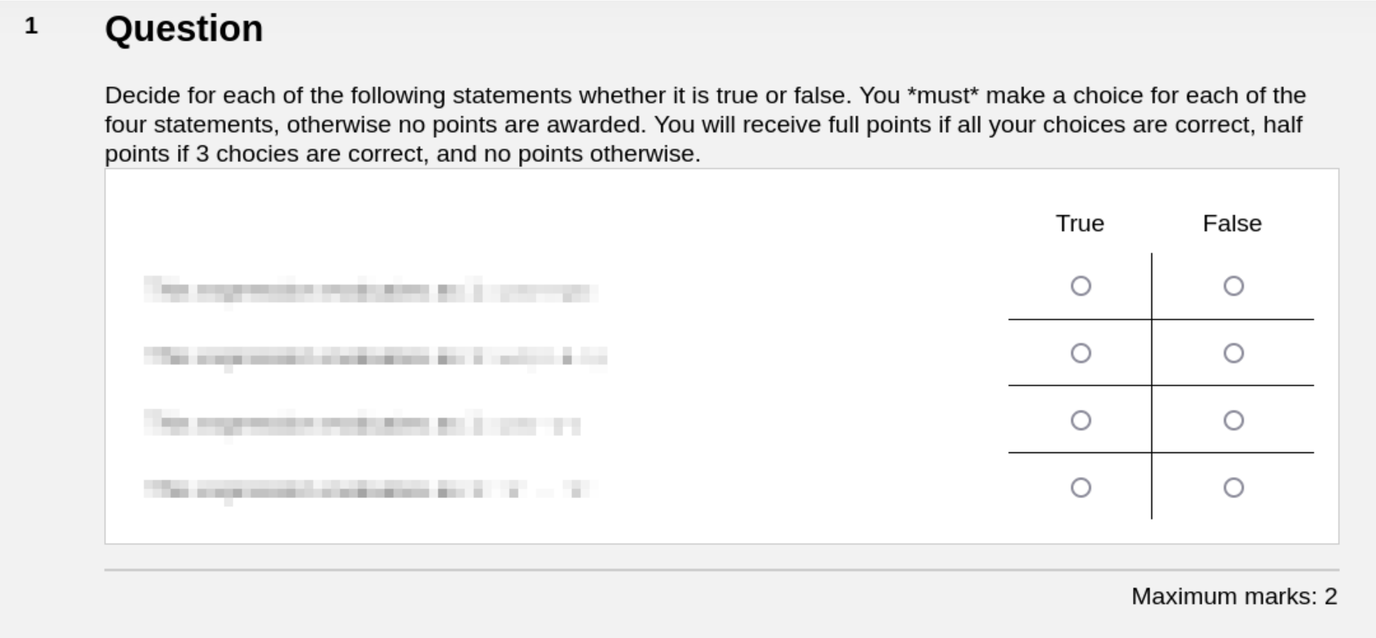
- You must be able to write source code on your own local machine (the choice of editors and/or IDE is yours).
- You are allowed to use your IDE and resources from the internet (Python API, search engines, StackOverflow, etc.). In the programming tasks, you may import functionality from the Python Standard Library.
- Absolutely forbidden is any kind of communication with any other people during the exam.

Exam structure

Section	Points
Multiple Choice (kprim)	2
Multiple Choice (kprim)	2
Programming	3
Programming	6
Programming	5
Programming	6
Programming	7
Programming	6
Programming	21
Multiple Choice (kprim)	2

- 60 Points total (1 point ≈ 1 minute)
- Many of the programming tasks are shockingly simple
- You **cannot** navigate forward and backward
  - You can go back to the beginning via 'go back to dashboard' at the end
- The exam is automatically submitted after 60min if you don't submit it yourself
- Answers are saved continuously

MC Questions



1 **Question**

Decide for each of the following statements whether it is true or false. You *must* make a choice for each of the four statements, otherwise no points are awarded. You will receive full points if all your choices are correct, half points if 3 chocies are correct, and no points otherwise.

	True	False
[illegible]	○	○
[illegible]	○	○
[illegible]	○	○
[illegible]	○	○

Maximum marks: 2

- 4 statements
- Each statement is True or False (True/False columns may be reversed!)
- 4 correct answers = 2 points
- 3 correct answers = 1 point
- 1 or 2 correct answers = 0 points

- Check code by running it in a python shell

Programming Tasks

- Examples are integral to the task description
  - Example calls
  - Example results
- Copy your solution into the answer field
  - Include anything that's in the *template*
  - Include signature (def...)
  - Do *not* include calls

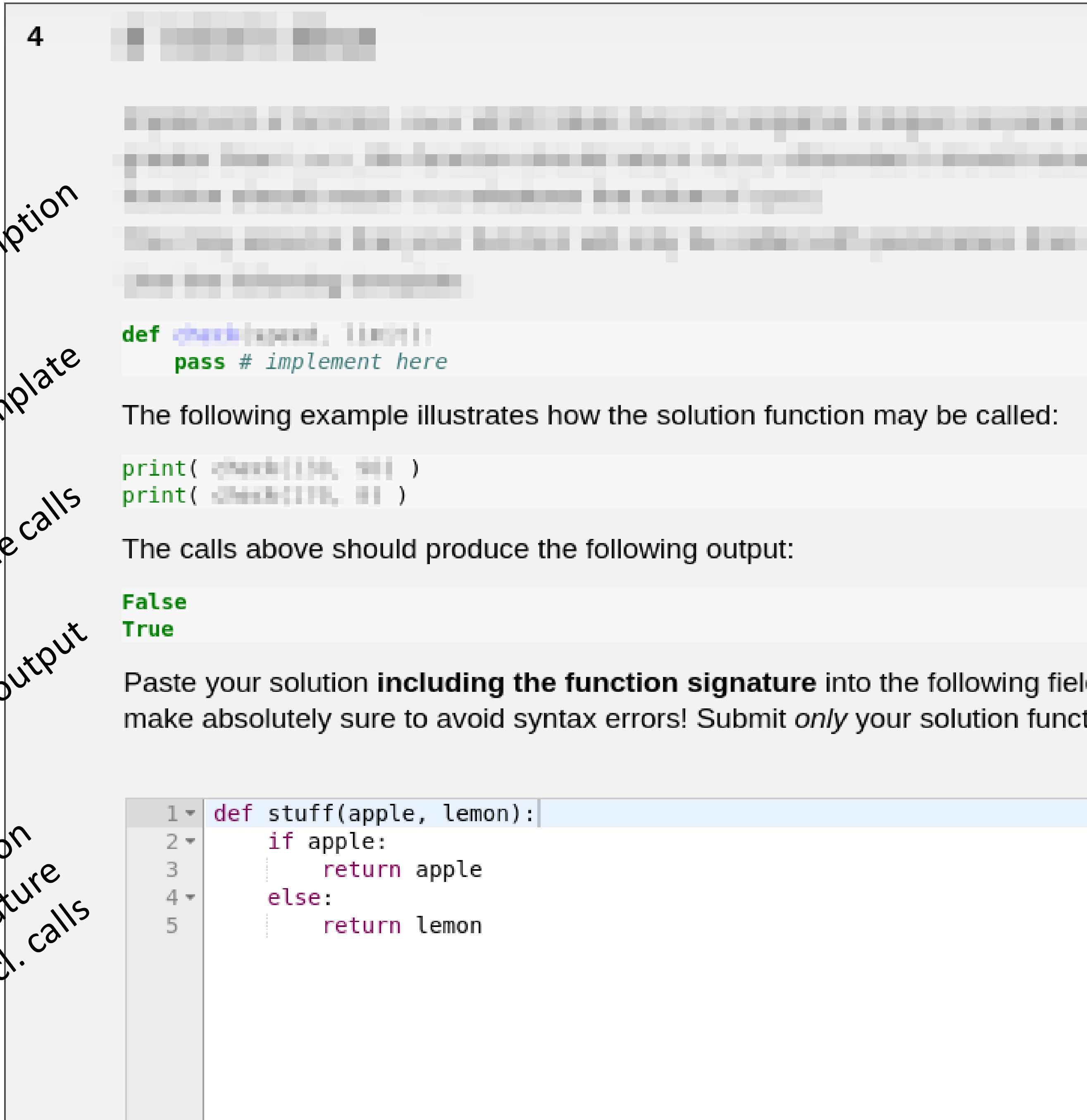
Task description

Code template

Example calls

Example output

You solution
inc. signature
excl. calls



4 [redacted]

[redacted]

```
def [redacted]:
    pass # implement here
```

The following example illustrates how the solution function may be called:

```
print( [redacted] )
print( [redacted] )
```

The calls above should produce the following output:

```
False
True
```

Paste your solution **including the function signature** into the following fiel
make absolutely sure to avoid syntax errors! Submit *only* your solution funct

```
def stuff(apple, lemon):
    if apple:
        return apple
    else:
        return lemon
```

Grading?

- Solutions with syntax errors will receive 0 points
  - More generally: solutions that crash when run will receive 0 points
- Similar to ACCESS: Points for fulfilling the specification
  - Partial points if the solution works for a subset of possible inputs
  - If a task asks for multiple return values, make sure you return the correct number of return values to earn partial points
- No penalty for skipping tasks.
- At the very least, make sure that the provided example calls produce the example output provided

How to solve a programming task

- Read the task carefully.
- **Make sure you understand it.** Don't make wild assumptions. If some instructions appear to be "missing", use an assumption that is most general and least complex. Look at the example calls for more info.
- Break down the problem into multiple sub-problems.
- Copy the template into your programming environment.
- Solve the task:
  - Run the code often
  - Print where you are unsure what happens
  - Test your solution with **all input cases** you can imagine

Help during the exam

- Microsoft Teams Channel "Info1 Exam Support 2022"
  - https://teams.microsoft.com/l/team/19%3akhkeM8m7v--19TaeFjNDHggs76CwNIBcrwYPSnH47NY1%40thread.tacv2/conversations?groupId=d9c96a55-05e9-4f47-86d7-2a21846fa766&tenantId=c7e438db-e462-4c22-a90a-c358b16980b3

- Only for technical problems, for example:
  - Exam/Task doesn't show up, can't go to next question, etc.

- NOT for content-related questions, for example:
  - "The task says to take a list of strings, what does this mean?"
  - Do NOT share exam content (questions, code)

Most importantly: practice programming!

- Programming is not a knowledge skill, but a crafting skill
- You learn by doing
- It requires practice
- Re-do old exercises without peeking at the solution.
- Solve old exams (Material folder on OLAT)
- Get used to Python by making mistakes and then fixing them
- Don’t give up on encountering errors, they always happen and solving them is an essential part of programming

Common Mistakes, Problem Solving

Dr. Carol Alexandru-Funakoshi

University of Zurich, Department of Informatics

print vs. return

- `print` and `return` are completely different things!
- `print`
  - Is a function that outputs a String to the console (usually)
  - Returns None
- `return`
  - Is a keyword
  - Returns a value of any type from a function (and ends the function)

```
>>> def say_my_name(name):
...     print(name)
...
>>> x = say_my_name("Bob")
Bob
>>> type(x)
<class 'NoneType'>
```

returns None

```
>>> def return_my_name(name):
...     return name
...
>>> y = return_my_name("Bob")
>>> type(y)
<class 'str'>
```

returns a String

Variable scoping

- Do NOT declare variables outside of your solution function
- Variables outside the function are in global scope
- If the solution function is called more than once, its behavior will be different!
- Do **not** use `global`!

```
s = 0
def mysum(l):
    global s
    for e in l:
        s += e
    return s

assert(mysum([1,2,3]) == 6) # passes
assert(mysum([1,2,3]) == 6) # fails, 12 != 6
```

Trouble with if/elif/else

- Every `if` starts a new condition expression!
- Use `elif` to continue an expression!

```
def numbername(x):
    res = ""
    if x == 1:
        res = "one"
    if x == 2:
        res = "two"
    if x == 3:
        res = "three"
    else:
        res = "I can only count to three"
    return res


print(numbername(2))
```

Trouble with if/elif/else

- Every `if` starts a new condition expression!
- Use `elif` to continue an expression!
- Or: be smart in how you use return

```
def numbername(x):
    if x == 1:
        return "one"
    if x == 2:
        return "two"
    if x == 3:
        return "three"
    return "I can only count to three"


print(numbername(2))
```

Returning **None** by accident

- If you're supposed to return a value, make sure you actually do in all cases!

```
def index_first_even(x):
    if x != []:
        for i, n in enumerate(x):
            if n % 2 == 0:
                return i
    else:
        return -1


print(index_first_even([1,2,3])) # 1
print(index_first_even([1,3,5])) # None
```

Returning **None** by accident

- If you're supposed to return a value, make sure you actually do in all cases!

- Solutions that *look* better are usually (but not always) better
- Simple solutions are often better than complicated ones
- Think before you implement!

```
def index_first_even(x):
    for i, n in enumerate(x):
        if n % 2 == 0:
            return i
    return -1


print(index_first_even([1,2,3])) # 1
print(index_first_even([1,3,5])) # -1
```

Important! Test all input cases!!!

- A quality assurance engineer walks into a bar.
  - orders a beer
  - orders 2 beers
  - orders 0 beers
  - orders 2^{2049} beers
  - orders -1 beers
  - orders 2.75 beers
  - orders a lizard
  - orders a agh)(!^%@_05"; drop table "account";--
  - orders a `\n\t\x00`

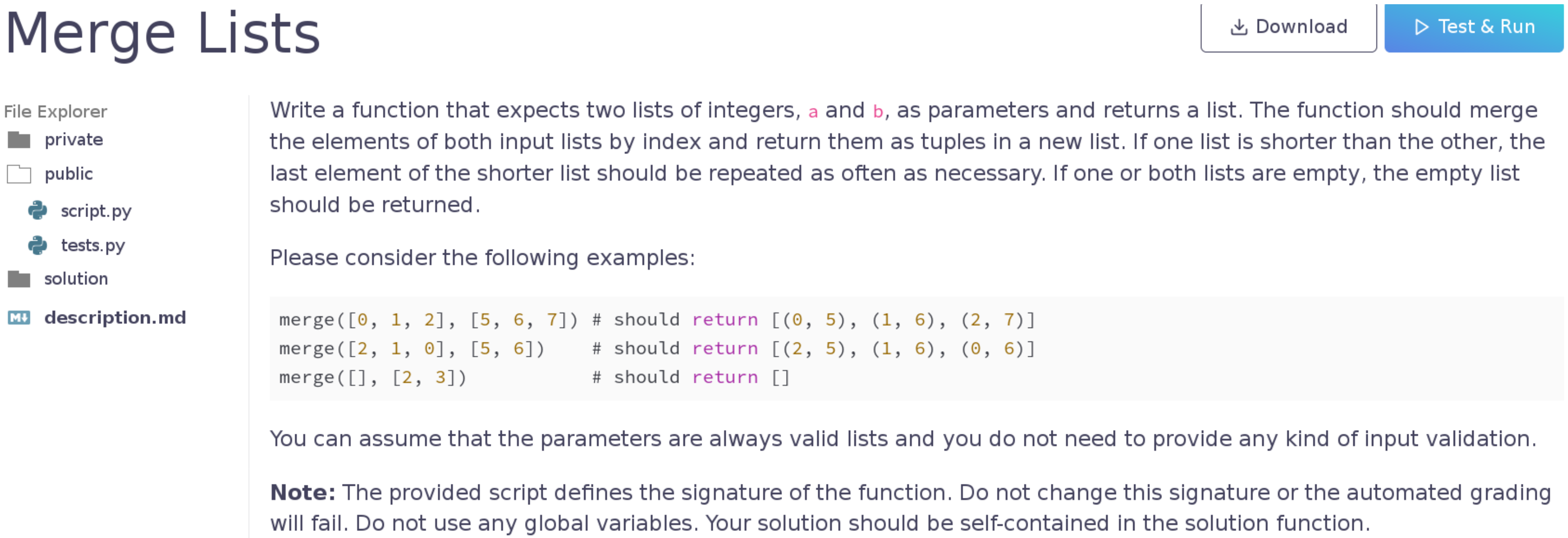
Important! Test all input cases!!!

- A quality assurance engineer walks into a bar.
  - orders a beer
  - orders 2 beers
  - orders 0 beers
  - orders 2^{2049} beers
  - orders -1 beers
  - orders 2.75 beers
  - orders a lizard
  - orders a agh)(!^%@_05"; drop table "account";--
  - orders a `\n\t\x00...`

...the first real customer walks in and asks where the bathroom is.

The bar bursts into flames, killing everyone.

Important! Test all input cases!!!



Merge Lists

Write a function that expects two lists of integers, `a` and `b`, as parameters and returns a list. The function should merge the elements of both input lists by index and return them as tuples in a new list. If one list is shorter than the other, the last element of the shorter list should be repeated as often as necessary. If one or both lists are empty, the empty list should be returned.

Please consider the following examples:

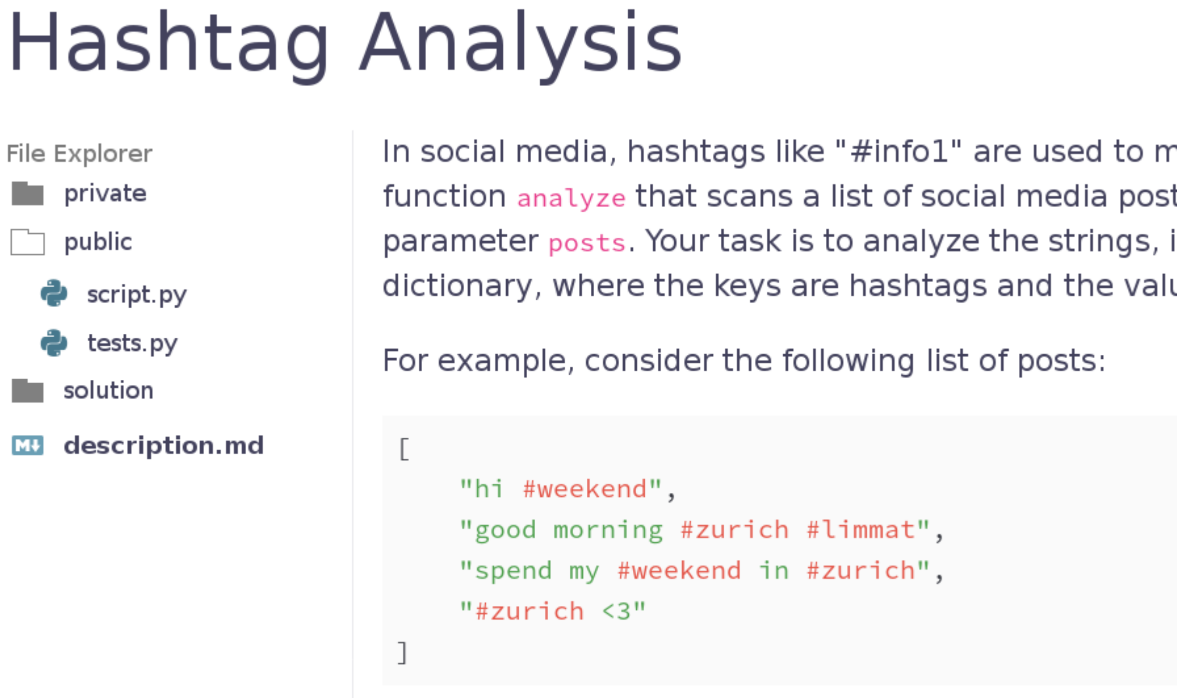
```
merge([0, 1, 2], [5, 6, 7]) # should return [(0, 5), (1, 6), (2, 7)]
merge([2, 1, 0], [5, 6])    # should return [(2, 5), (1, 6), (0, 6)]
merge([], [2, 3])           # should return []
```

You can assume that the parameters are always valid lists and you do not need to provide any kind of input validation.

Note: The provided script defines the signature of the function. Do not change this signature or the automated grading will fail. Do not use any global variables. Your solution should be self-contained in the solution function.

- What to test?
  - a empty, b not empty
  - b empty, a not empty
  - a and b empty
  - a and be same non-zero length
  - a shorter than b
  - b shorter than a

Important! Test all input cases!!!



Hashtag Analysis

File Explorer
- private
- public
  - script.py
  - tests.py
- solution
- **description.md**

In social media, hashtags like "#info1" are used to m
function `analyze` that scans a list of social media post
parameter `posts`. Your task is to analyze the strings, i
dictionary, where the keys are hashtags and the valu

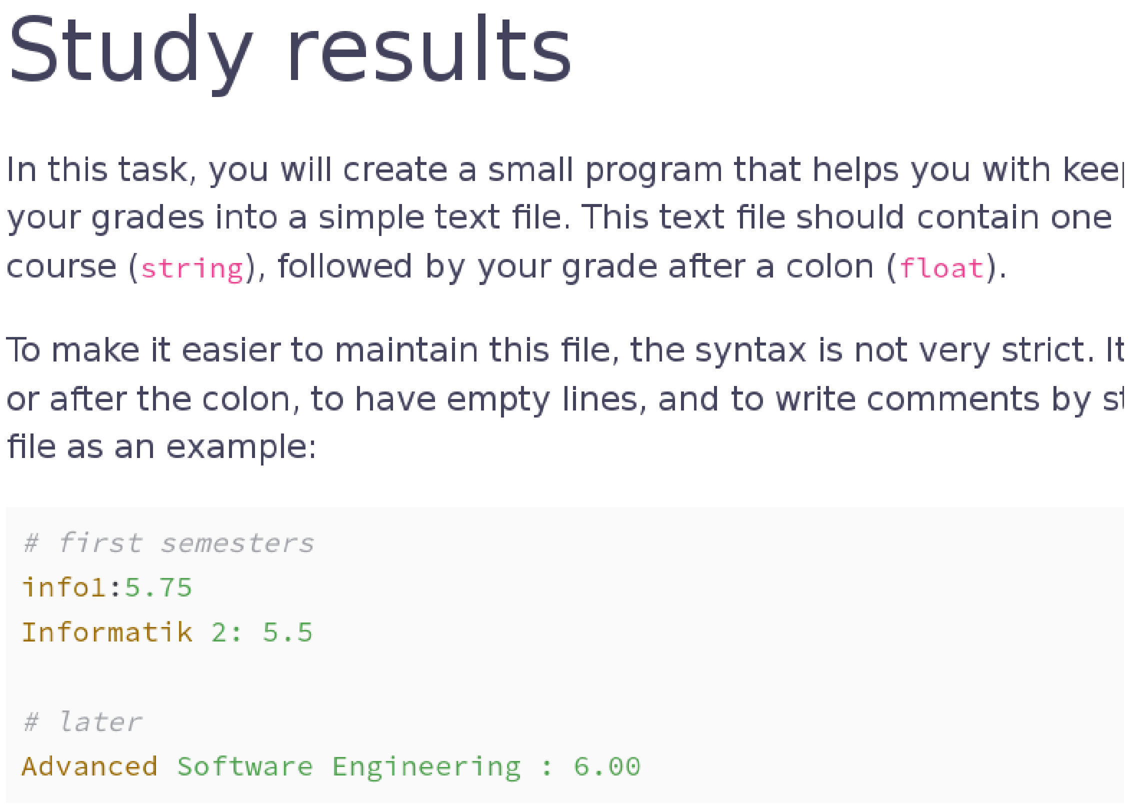
For example, consider the following list of posts:

```
[
    "hi #weekend",
    "good morning #zurich #limmat",
    "spend my #weekend in #zurich",
    "#zurich <3"
]
```

- What to test?
  - `[ "", "#", "##", "###", "#-", "#1",`
    `"#a", "#b#c", "#d #e", " #f", "g#h", "#ii#jjjj",`
    `".#k", ".#l.", "--#m--", "##n", "-#o ", "#p-", " #q "]`

Important! Test all input cases!!!

- What to test?
  - Empty file
    - Totally empty
    - Only blank lines
    - Only blank and comment lines
    - Only comment lines
  - Non-existent file
  - "abc:5"
  - "abc:5.0"
  - "abc   :5"
  - "abc: 5"
  - "abc   :   5"
  - "abc\t\t:5"
  - "abc:\t\t5"
  - "abc\t\t:\t\t5"



Study results

In this task, you will create a small program that helps you with kee
your grades into a simple text file. This text file should contain one
course (`string`), followed by your grade after a colon (`float`).

To make it easier to maintain this file, the syntax is not very strict. It
or after the colon, to have empty lines, and to write comments by st
file as an example:

```
# first semesters
info1:5.75
Informatik 2: 5.5

# later
Advanced Software Engineering : 6.00
```

Review this!

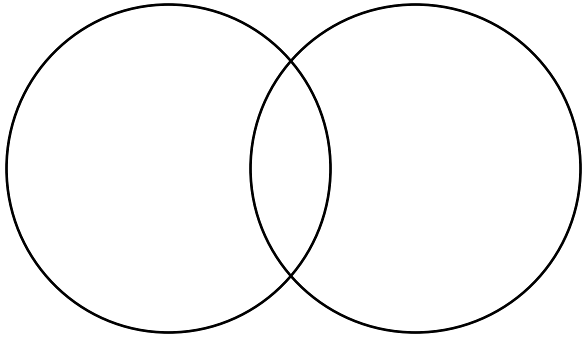
- You must know how functions work in Python!
  - A function has zero or more parameters; some could be optional
  - Know how to call a function
  - Know that functions can be passed as parameters and returned
  - Global variables vs. parameters
- Know how to use lists, tuples, dicts
  - Know about enumerate, .values(), .items(), x in y, etc...
- Know about string manipulation (find(), split(), strip(), join(), etc.)

A few more things...

- Using the `global` keyword is 100% forbidden.
  - You shouldn't be specifying variables outside your solution function anyway
- Read the task carefully
  - "a list", "a tuple" or "a dictionary" implies that these could be empty. Otherwise, the task would explicitly say "a non-empty list", "a non-empty dictionary", etc...
  - Mind terms such as "non-negative", "positive", "integer", "number", etc.
- Prepare your environment!
  - IDE ready? Most important Python documentation API docs open? Lecture slides at hand? Battery full or plugged into wall? Stable internet connection?

Sets can be useful

- A set is an unordered collection of unique values
- You can construct a set from a list or tuple by calling set() on it.
- Supports operations like add(), remove(), &, |, -, ^



```
>>> x = {1, 2, 3, 4, 4}
>>> y = set([3, 4, 5])
>>> x
{1, 2, 3, 4}
>>> y
{3, 4, 5}
>>> x & y
{3, 4}
>>> x | y
{1, 2, 3, 4, 5}
>>> x - y
{1, 2}
>>> x ^ y
{1, 2, 5}
```

Random seed

- Sometimes you want randomness to be "predictable".
- By setting a specific "random seed", random values will always be produced in the same sequence.
- This can be useful for testing, because it makes functions using randomness reproducible

```
>>> import random
>>> random.randint(0,10)
3
>>> random.randint(0,10)
4
>>> random.seed(1)
>>> random.randint(0,10)
2
>>> random.randint(0,10)
9
>>> random.seed(1)
>>> random.randint(0,10)
2
>>> random.randint(0,10)
9
```

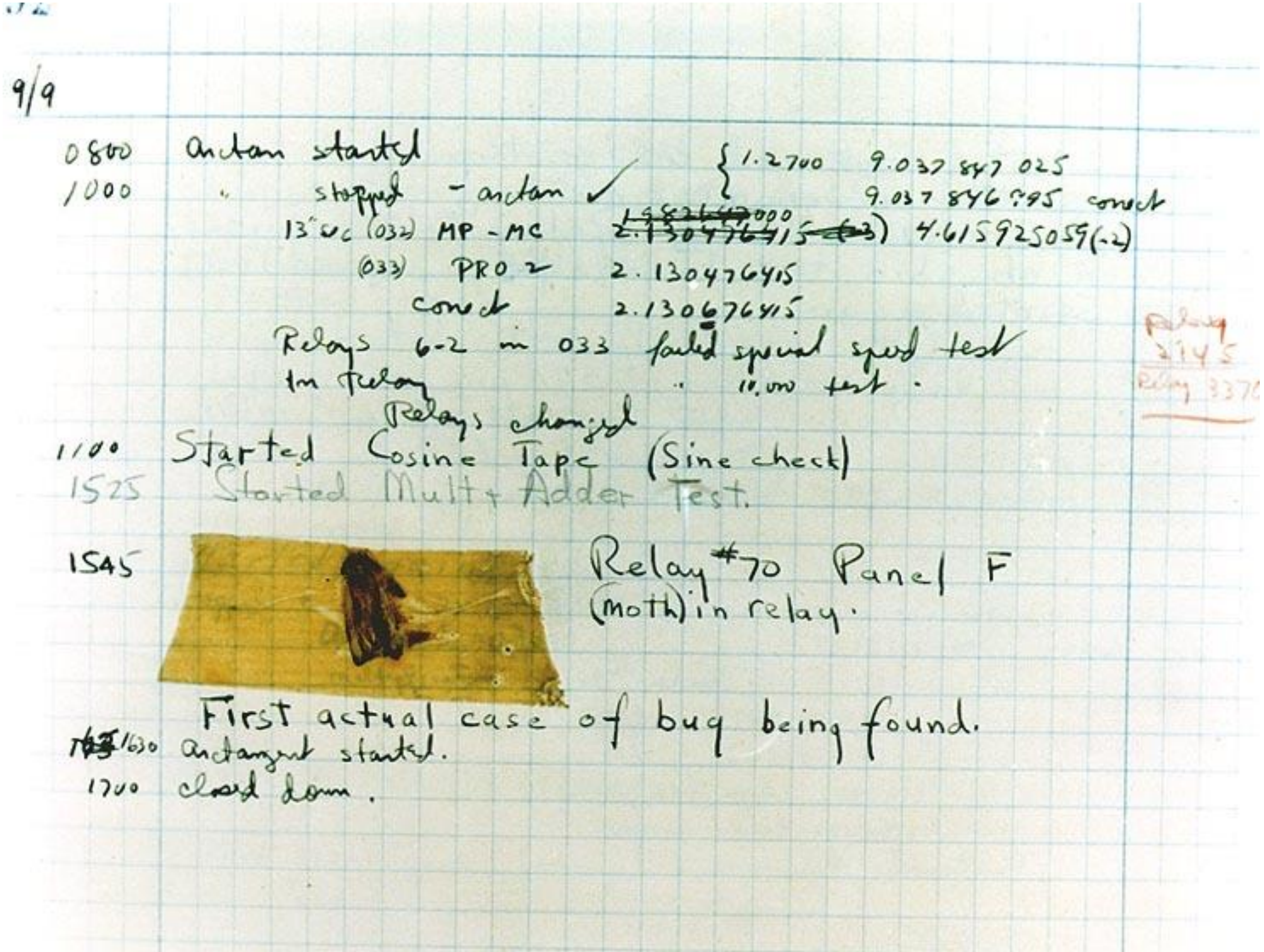
Questions about the exam?

Control Flow, Debugging, and Testing

Dr. Carol Alexandru-Funakoshi

University of Zurich, Department of Informatics

What is a bug?



9/9

0800 Antan started

1000 " stopped - antan ✓ {1.2700 9.037 847 025

9.037 846 995 conect

13" sc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ ~~(-3)~~ 4.615925059(-2)

(033) PRO 2 2.130476415

conect 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 11.000 test .

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

1630 Antangent started.

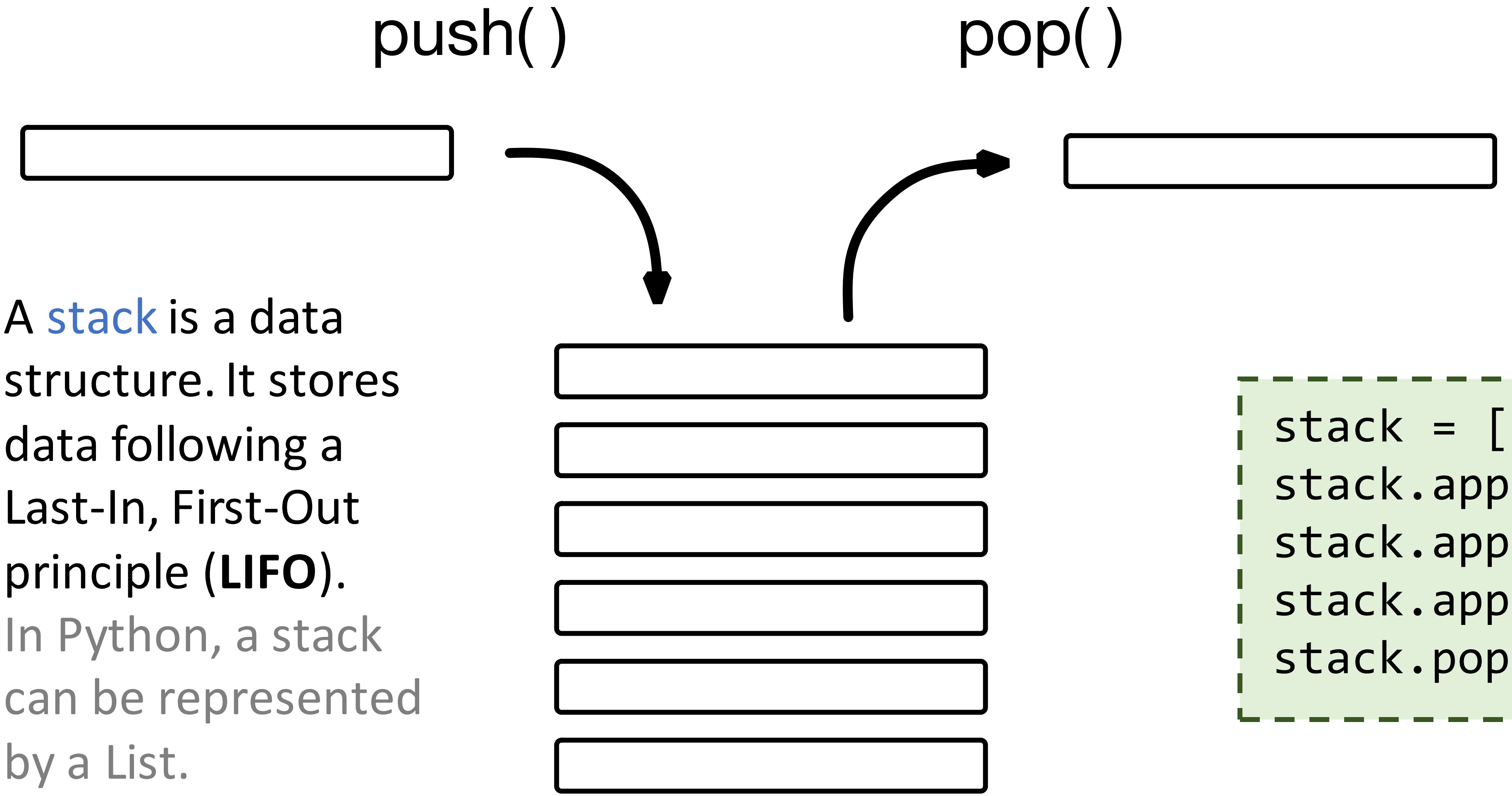
1700 closed down.

Relay 2145

Relay 3370

The Call Stack

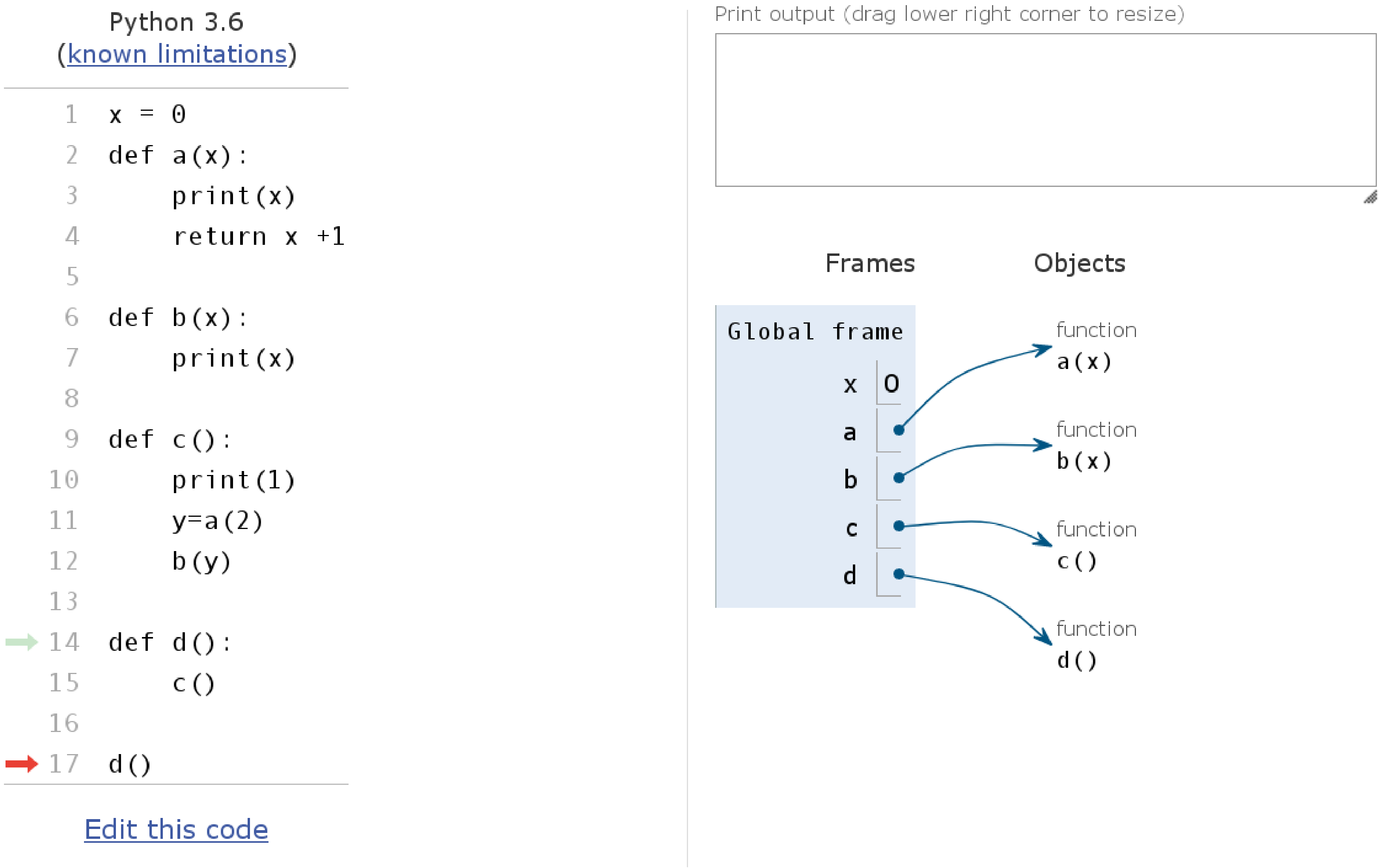
The "Stack" as a generic concept



A stack is a data structure. It stores data following a Last-In, First-Out principle (**LIFO**). In Python, a stack can be represented by a List.

```
stack = []
stack.append(3)
stack.append(17)
stack.append(6)
stack.pop() # 6
```

"Call frames" and local variables live on the call stack



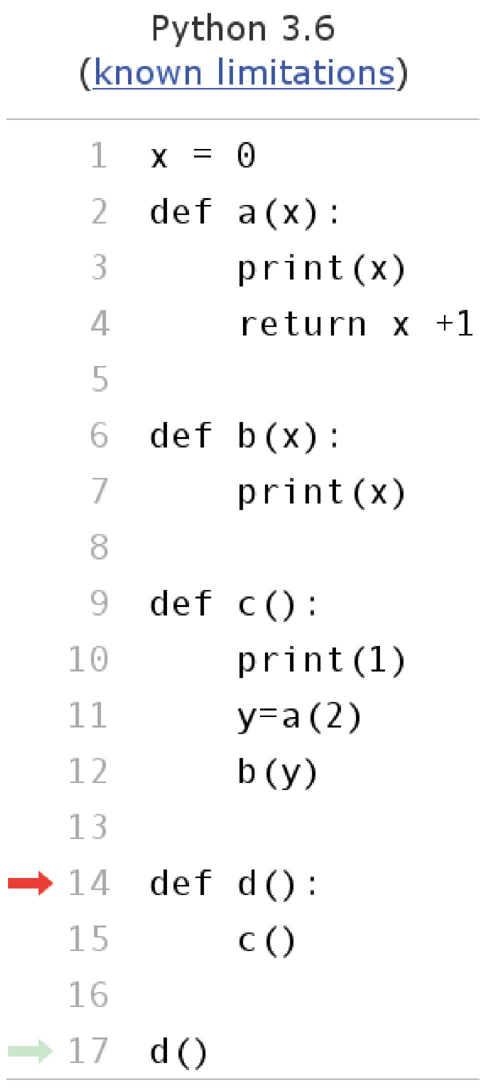
Python 3.6
(known limitations)

```
x = 0
def a(x):
    print(x)
    return x +1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()

d()
```

Edit this code

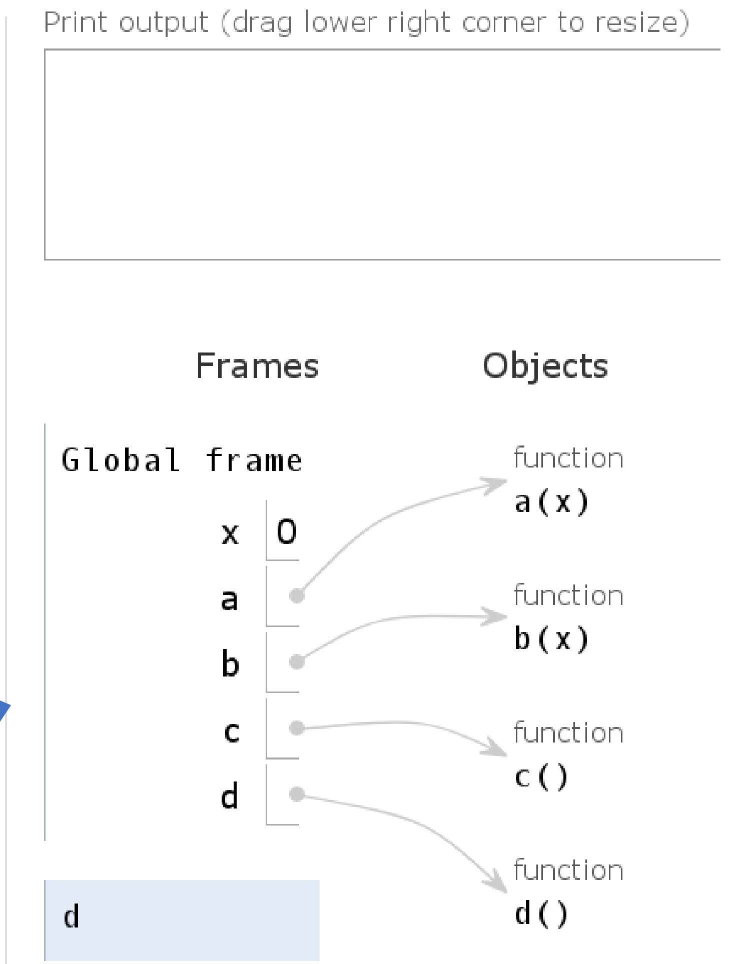
```
x = 0
def a(x):
    print(x)
    return x + 1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()
d()

# https://is.gd/VydBgy
```

"Call frames" and local variables live on the call stack



Python 3.6
(known limitations)

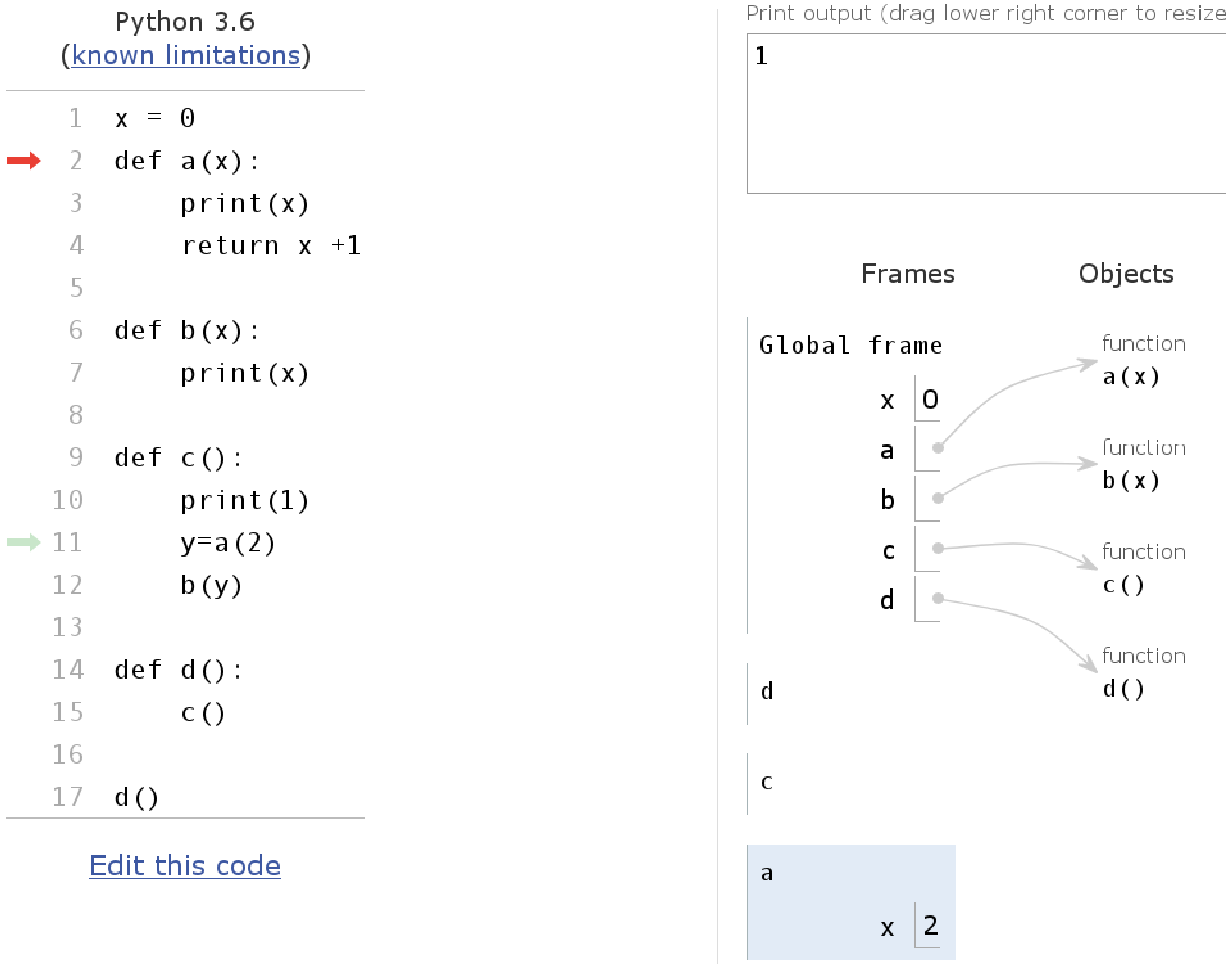
```
x = 0
def a(x):
    print(x)
    return x +1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()

d()
```



The stack "upside down"
The latest call is at the bottom, the oldest at the top

```
x = 0
def a(x):
    print(x)
    return x + 1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()
d()

# https://is.gd/VydBgy
```

"Call frames" and local variables live on the call stack



Python 3.6
(known limitations)

```
x = 0
def a(x):
    print(x)
    return x +1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()

d()
```

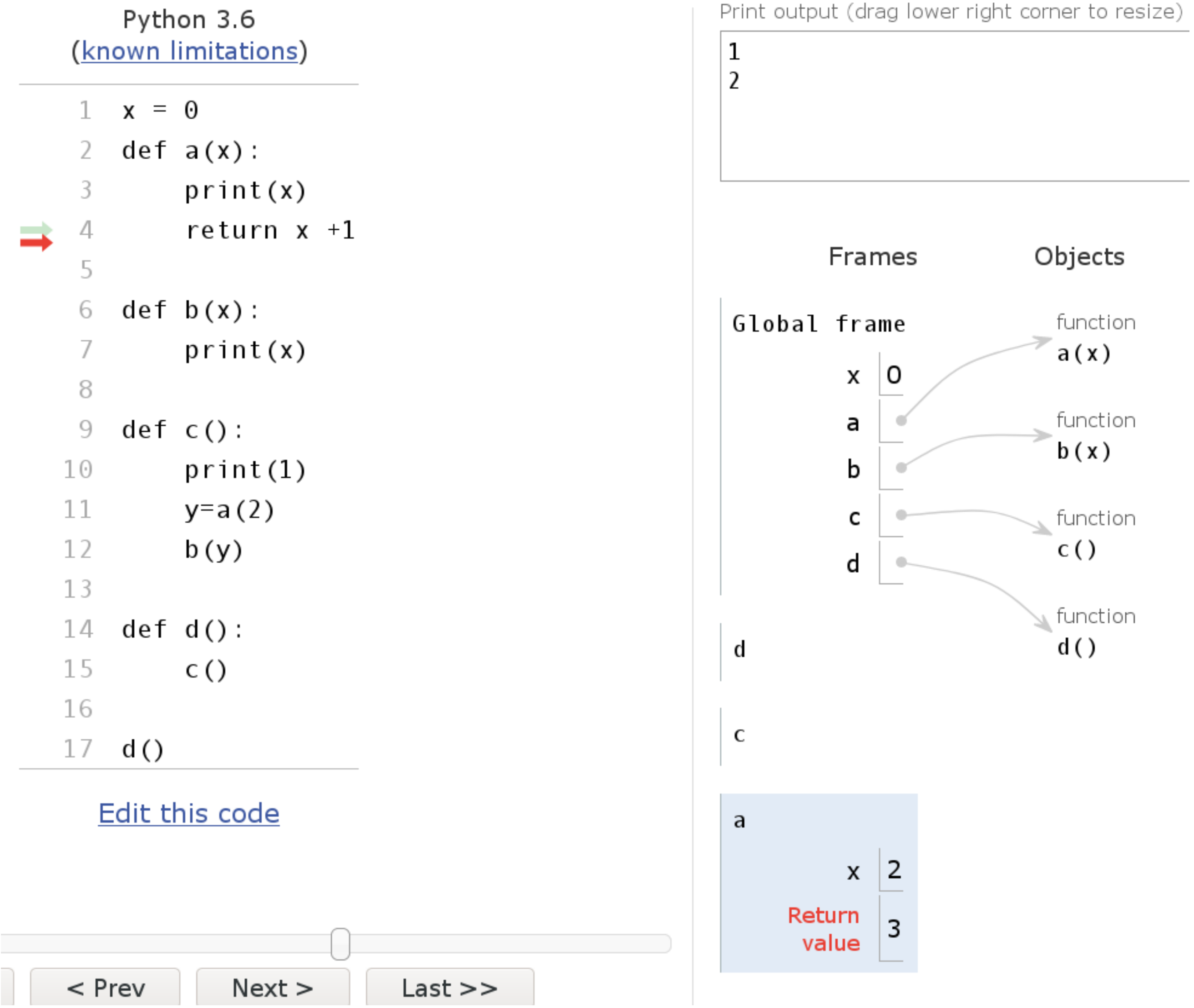
Edit this code

Print output (drag lower right corner to resize

```
1
```

```
x = 0
def a(x):
    print(x)
    return x + 1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()
d()

# https://is.gd/VydBgy
```

"Call frames" and local variables live on the call stack



```
x = 0
def a(x):
    print(x)
    return x +1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()

d()
```

```
1
2
```

Global frame

x	0
a	function a(x)
b	function b(x)
c	function c()
d	function d()

d

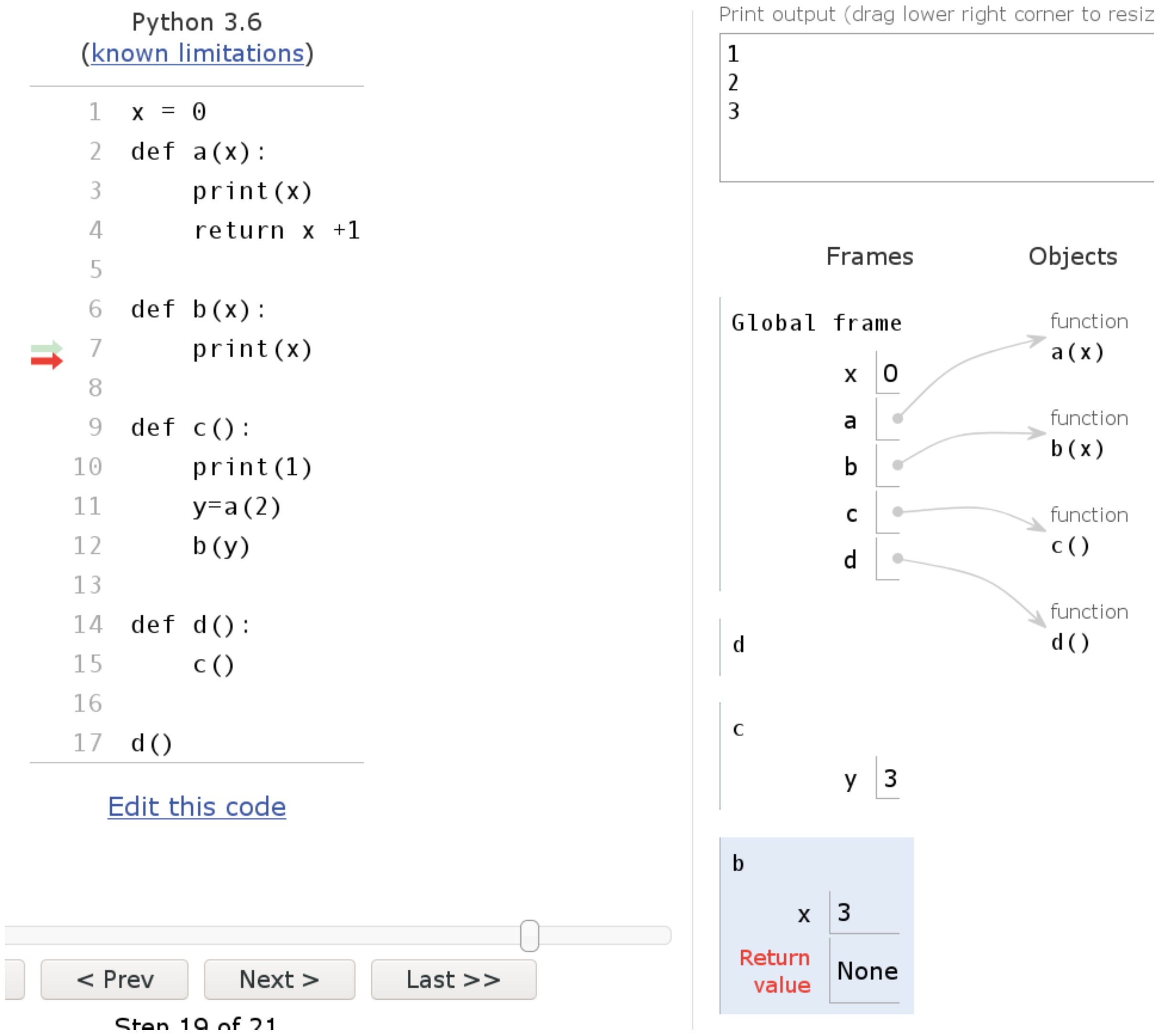
c

a

x	2
Return value	3

```
x = 0
def a(x):
    print(x)
    return x + 1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()
d()

# https://is.gd/VydBgy
```

"Call frames" and local variables live on the call stack



Python 3.6
(known limitations)

```
x = 0
def a(x):
    print(x)
    return x +1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()

d()
```

Edit this code

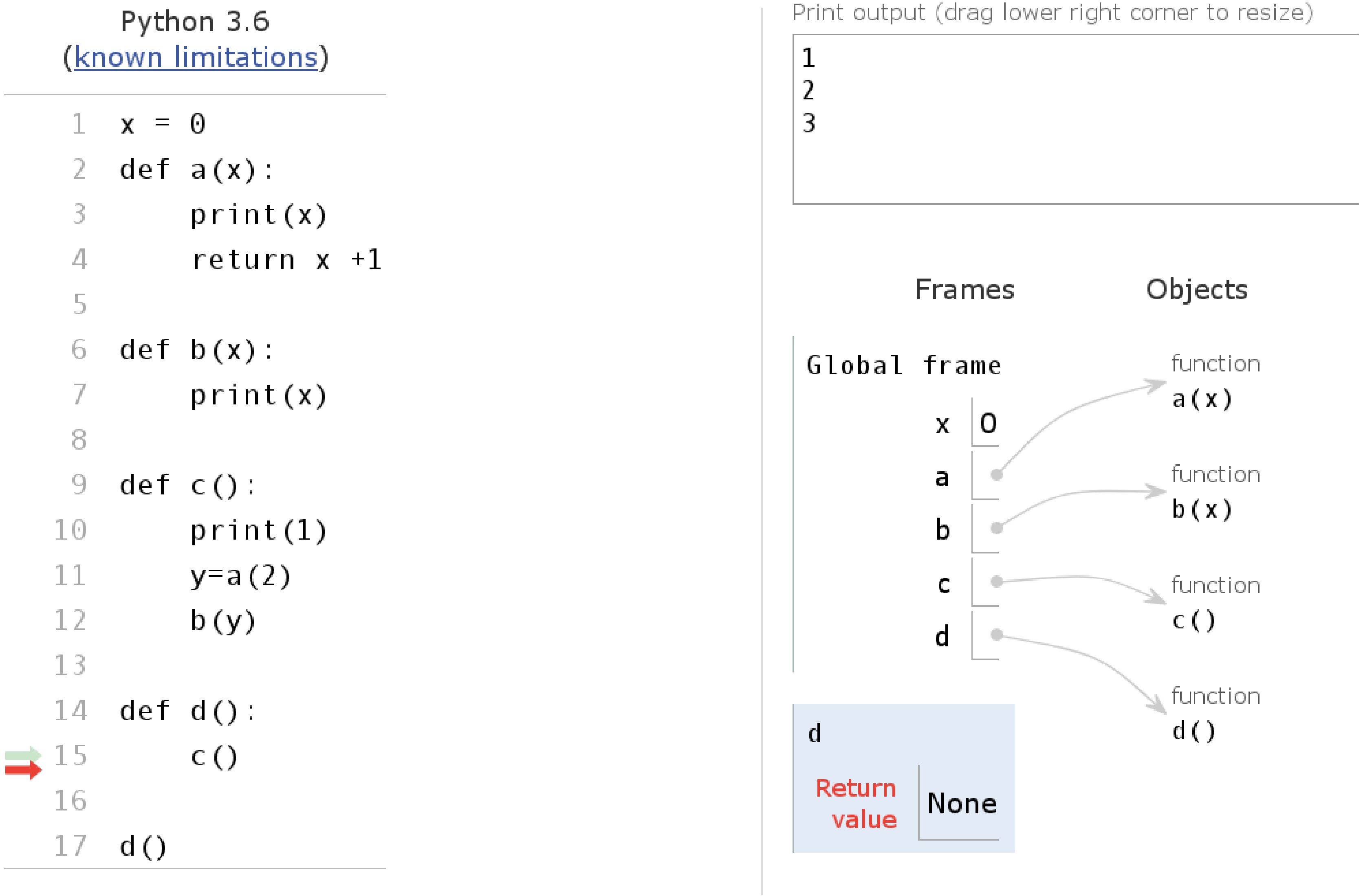
Step 19 of 21

Print output (drag lower right corner to resiz

```
1
2
3
```

```
x = 0
def a(x):
    print(x)
    return x + 1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()
d()

# https://is.gd/VydBgy
```

"Call frames" and local variables live on the call stack



Python 3.6
(known limitations)

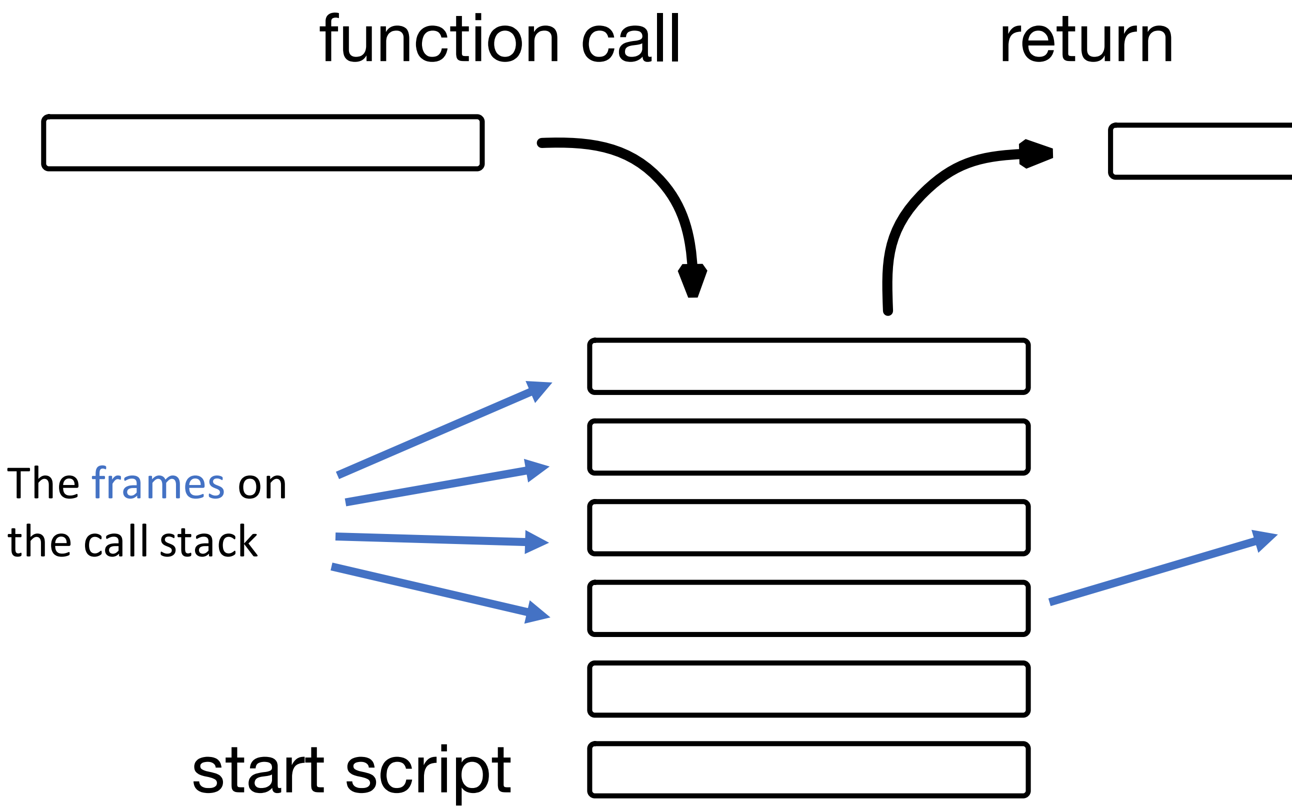
```
x = 0
def a(x):
    print(x)
    return x +1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()

d()
```

Print output (drag lower right corner to resize)

```
1
2
3
```

```
x = 0
def a(x):
    print(x)
    return x + 1

def b(x):
    print(x)

def c():
    print(1)
    y=a(2)
    b(y)

def d():
    c()
d()

# https://is.gd/VydBgy
```

The call stack



function call

return

The frames on the call stack

start script

Function: foo() in algorithm.py line 10

Variables:

Name	Type	Value
x	int	3
y	int	4
z	float	2.5

A "Stack Trace"

```
$ python callstack.py
1
2
Traceback (most recent call last):
  File "callstack.py", line 17, in <module>
    d()
  File "callstack.py", line 15, in d
    c()
  File "callstack.py", line 11, in c
    y=a(2)
  File "callstack.py", line 4, in a
    return x + 1 + "!"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
 1 x = 0
 2 def a(x):
 3     print(x)
 4     return x + 1 + "!"
 5
 6 def b(x):
 7     print(x)
 8
 9 def c():
10     print(1)
11     y=a(2)
12     b(y)
13
14 def d():
15    c()
16
17 d()
18
19 # https://is.gd/YUjxH8
```

The stack trace is "upside down":
The latest call is at the bottom, the oldest at the top

Error Handling / Exceptions

`try/except/else/finally`

Error Handling

- So far, we have always assumed that everything works as expected
- Programs often have to deal with unexpected conditions (e.g., invalid input, non-existing files, etc.)
- How can we handle errors?
- Several options:
  - Fail silently (Bad! Caller does not know that an error has occurred)
    - Python actually loves doing this, some languages are *stricter* than others
  - Return an error value (Bad! Caller has to check for several error conditions)
  - Raise an exception ("error or unexpected condition in the program")

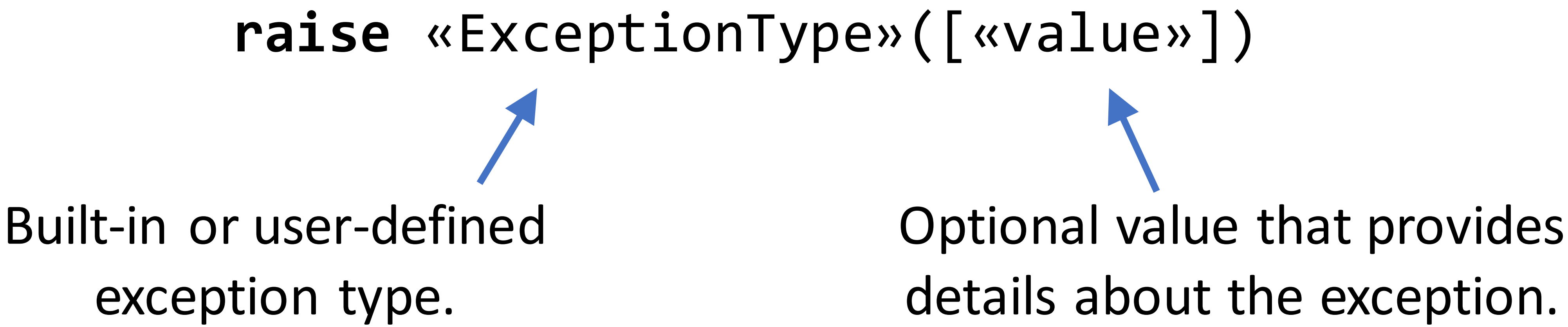
Exceptions raised by Python – Examples

```
l = [1, 2, 3]
l[10]       # IndexError: list index out of range

int(“foo”) # ValueError: invalid literal for int()
                      with base 10: 'not a number'

print(xxx) # NameError: name xxx' is not defined

10/0        # ZeroDivisionError: division by zero
```

Other Python Exceptions

- Built-in Exceptions
  - SyntaxError: the Python parser encountered an error
  - KeyError: a dictionary key is not found
  - FileNotFoundError: trying to open a file or directory that does not exist
  - ValueError: raised for inappropriate values
  - ...
  - find more at https://docs.python.org/3/library/exceptions.html
- It is also possible to introduce user-defined exceptions

Raising Exceptions



```
raise «ExceptionType»([«value»])
```

Built-in or user-defined exception type.

Optional value that provides details about the exception.

```
raise ValueError(“Number too small”)
```

It's called `raise` because it "raises" the error to the parent call frame

Example: Checking Parameters

```
def do_something_with_number(n):
    if n < 0:
        raise ValueError(“Need positive number”)
    ...
```

Remember, functions define a "contract", i.e., the expectations they have towards the input and the guarantees for the output. Such a check can be used to enforce these assumptions and to provide meaningful feedback in case of violations.

Handling Exceptions

Code that can cause an exception is placed in a try block.

```
try:
    # see previous slide
    do_something_with_number(-1)
except:
    print(“Ooops, there was an error!”)

print(“Life must go on.”)
```

The except block specifies how the error is handled instead of crashing with a `ValueError`.

In any case, execution continues after the try block

Handling specific exceptions

```
a = input(“Please enter a number: ”)
b = input(“Please enter another number: ”)


div = float(a) / float(b)
print(“{} / {} = {}”.format(a, b, div))
 # what if the user doesn't enter numbers?
 # what if the user enters 0 for b?
```

try/except/else/finally Syntax

```
try:                           # <- Try what's in this block
    print("try1")              #    Here goes the normal behavior
    fun_that_might_raise_ex()
    print("try2")
except ZeroDivisionError:      # <- What to do if this specific
    print("specific except")   #    exception is thrown
except:                        # <- What to do if any other
    print("catch-all except")  #    exception is thrown
else:                          # <- In case no exceptions
    print("else")              #    were thrown
finally:                       # <- Always run this block, whether
    print("finally")           #    there were exceptions or not
```

Handling specific exceptions

```
a = input(“Please enter a number: ”)
b = input(“Please enter another number: ”)

try:
    div = float(a) / float(b)
    print(“{} / {} = {}”.format(a, b, div))
except ValueError:
    print(“Please enter two numbers!”)
except ZeroDivisionError:
    print(“Second number must not be 0!”)
```

Valid Syntax

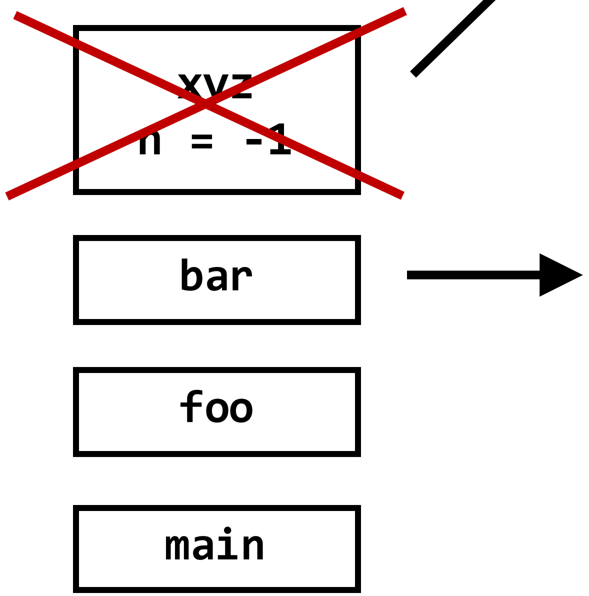
```
try:
    ...
except:
    ...
```

```
try:
    ...
finally:
    ...
```

A `try` cannot stand alone. A minimal `try` needs either some `except` block or a `finally` block.

- `else` is optional
- `finally` is optional if there's an `except`
- `except` is optional if there's a `finally`
- Having more than one `except` is optional

What happens to the stack when an exception is raised?



```
def xyz(n):
    if n < 0:
        raise ValueError(f"bad value: {n}")
    ...
```

```
def bar():
    try:
        xyz(-1)
    except ValueError as e:
        print(“Error: ”, e.args)
```

The actual exception object can be bound to a name to access its contents.

Name	Type	Value
args	tuple	("bad value: -1",)

Exceptions summarized

- Need at least `try/except` or `try/finally`
- You may think of exception raising as a "`return` for errors"
    - Don't use `return` to indicate failure or errors, but for "normal" data flow
        - E.g.: Get *a list of records* for *a specific person*:
            - Person does not exist: `raise` an exception
            - Person has no records: `return` an empty list

```
def get_records(person):
  """Returns a list of records for person"""
  ...
```

-
    - Use `return` for "normal" program logic, use `raise` for error cases
- Avoid "catch-all" `excepts`, they are bad style.
    - Catch only those errors that you intend to handle

Example 1

- You are writing a chat bot that can help with all kinds of clothing-related questions. You are using many different APIs for this. One provides the size label for a given chest circumference.
- The existing API returns a string label for a given circumference x
  - If x is out of known range a `NotImplementedError` is raised.
  - A `ValueError` is raised if x does not contain a number.
- Create a function prompt() which
  - Asks for input and passes the input to get_label
  - If a `NotImplementedError` is raised, we make custom clothing
  - If a `ValueError` is raised, ask again for input

```
def get_label(x):
    import numbers
    if not isinstance(x, numbers.Number):
        try:
            x = float(x)
        except ValueError:
            raise ValueError(
                  "Input is not a number")
    if x <= 80 or x > 124:
        raise NotImplementedError
    if x <= 90:
        return "XS"
    elif x <= 98:
        return "S"
    elif x <= 104:
        return "M"
    elif x <= 111:
        return "L"
    else:
        return "XL"
```

Example 1 – Solution

```
from label import get_label

def prompt():
    answer = ""
    while not answer:
        query = input("What size do you need? ")
        try:
            answer = f"We have {get_label(query)} in store!"
        except ValueError:
            print("Please enter a number :)")
        except NotImplementedError:
            answer = "We will custom-make your size!"
    print(answer)
```

Debugging

How can we find bugs in our code?

- So far we know:
  - Testing different inputs, see if it works correctly (black-box testing)
  - Littering the code with `print` statements to keep track of what's happening
  - Stepping through the execution "in our heads"

How can we find bugs in our code?

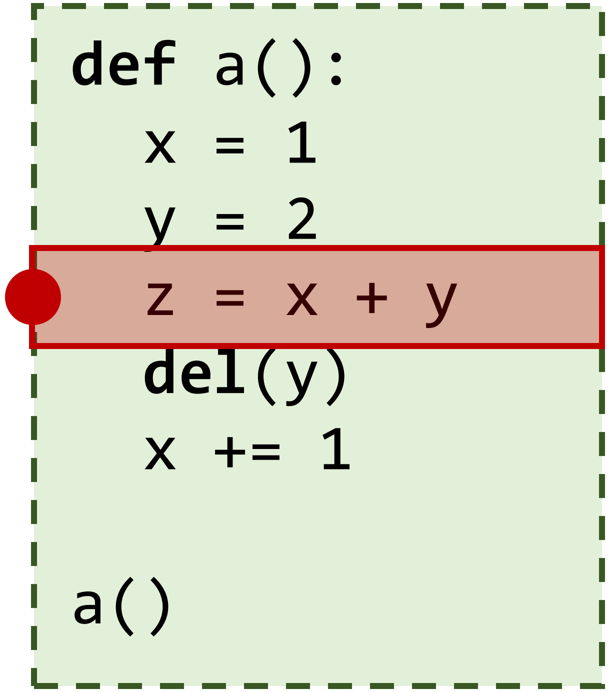
- So far we know:
  - Testing different inputs, see if it works correctly (black-box testing)
  - Littering the code with `print` statements to keep track of what's happening
  - Stepping through the execution "in our heads"

- There's one more sophisticated tool: the "debugger"
  - Allows executing code one step at a time (the program is paused)
  - Allows setting "break points" to skip ahead to a pause
  - Allows showing the call stack, incl. local variables on the call stack

● Breakpoints
⤓ Step Over
↘ Step Into
↗ Step Out
▶ Continue
■ Abort

Breakpoints

"Stop here during debugging execution"



```
def a():
   x = 1
   y = 2
   z = x + y
   del(y)
   x += 1

a()
```

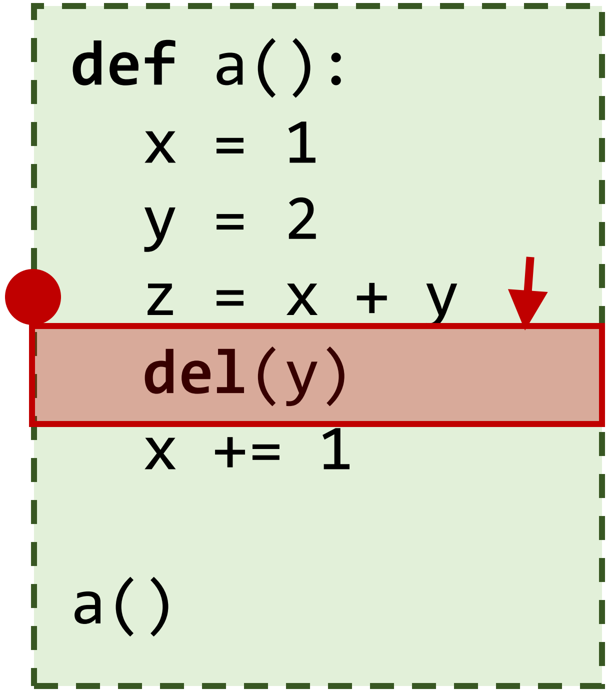
Name	Type	Value
x	int	1
y	int	2

Variable Inspector

Select a line (statement) in the code where the execution should pause

Step Over

"Go to the next statement on the same level"



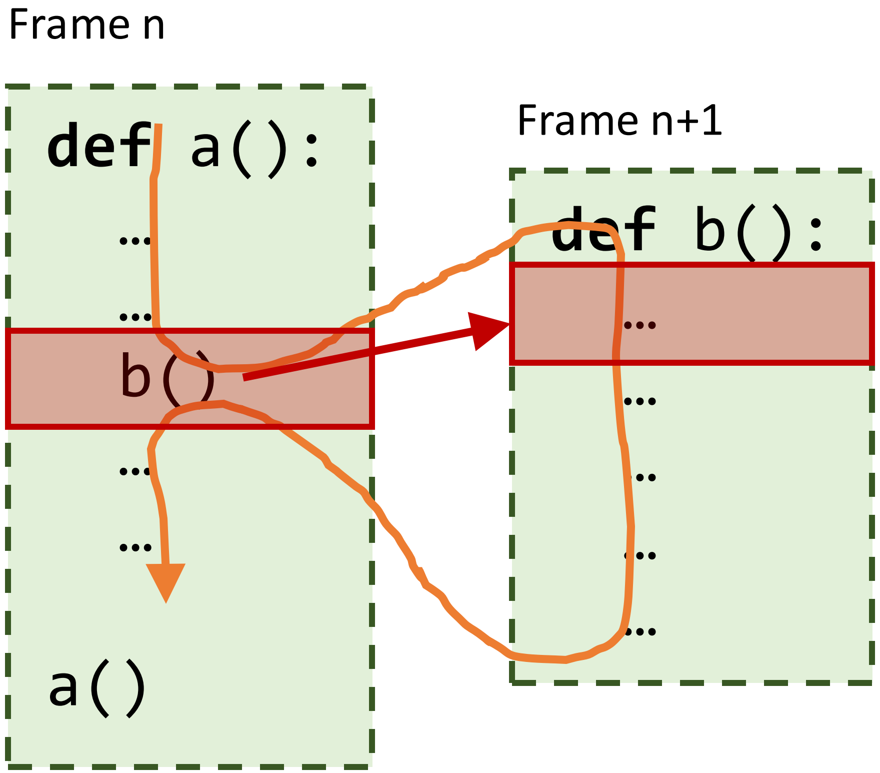
```
def a():
   x = 1
   y = 2
   z = x + y
   del(y)
   x += 1

a()
```

Name	Type	Value
x	int	1
y	int	2
z	int	3

The current statement will be executed and the execution stops at the next statement in the same function.

Step Into

“Follow the control-flow into this function call”

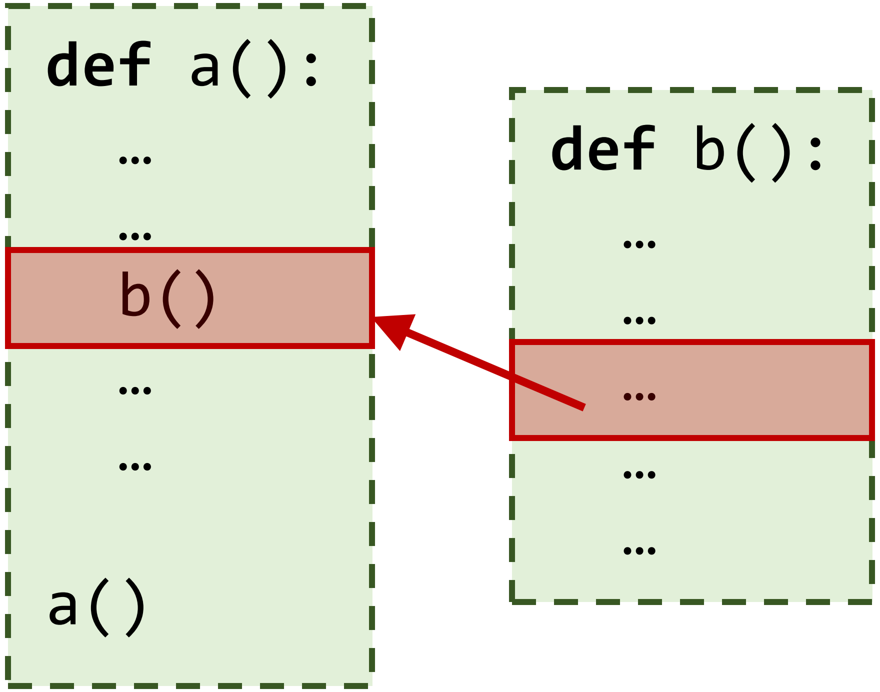


The current statement could be a call, which adds another stack frame. The debugger will *step into* this call and stop at the first statement in the new function.

For primitive operations, like additions, a *step into* behaves like a *step over*.

Step Out

"Execute the rest of this function and then stop in the caller"



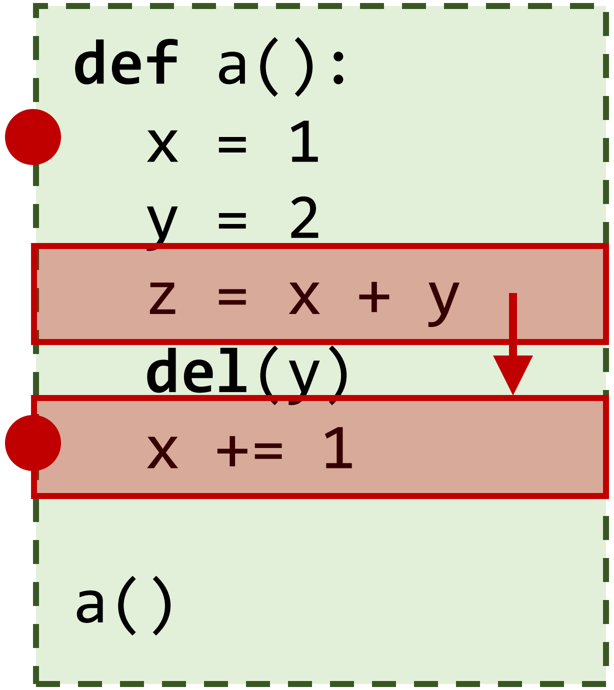
```
def a():
    …
    …
    b()
    …
    …

a()
```

```
def b():
    …
    …
    …
    …
    …
```

A *step out* finishes the *debugging* of the current function. The *program* will continue, and the *debugger* will stop again when the current stack frame has been removed from the stack.

Continue

“Continue with regular program execution (until the next breakpoint)”



```
def a():
    x = 1
    y = 2
    z = x + y
    del(y)
    x += 1

a()
```

A *continue* stops the stepwise execution of the program. The execution will continue normally until the next breakpoint is hit or until the program terminates.

Abort

“Abort the current debugging session / program execution”

An *abort* stops the execution of the program. The remaining statements will not be executed anymore.

Debugging

- ● Breakpoints
- ⤓ Step Over
- ↘ Step Into
- ↗ Step Out
- ▶ Continue
- ■ Abort

- **Debugging basics:**
  - Step over/into, set breakpoints, abort/continue
  - Available in any common IDE
  - Available on the command line or as an API using pdb: https://docs.python.org/3/library/pdb.html
- **Advanced Debugging**
  - Set "Watch Expressions" (see the value of a particular expression)
  - Change Values On-the-fly (change contents of variables)
  - Conditional Breakpoints (breakpoints only hit under certain conditions)
- A debugger can be very useful, *but slow*, process for tricky bugs that are hard to track down, because you can step through the program execution step by step.

You should be able to answer the following:

- What is a call stack? A Frame?
- What are exceptions? How do you raise and handle them?
- How can I stop a program execution at a specific point?
- Which tools do I have to interact with a running program?