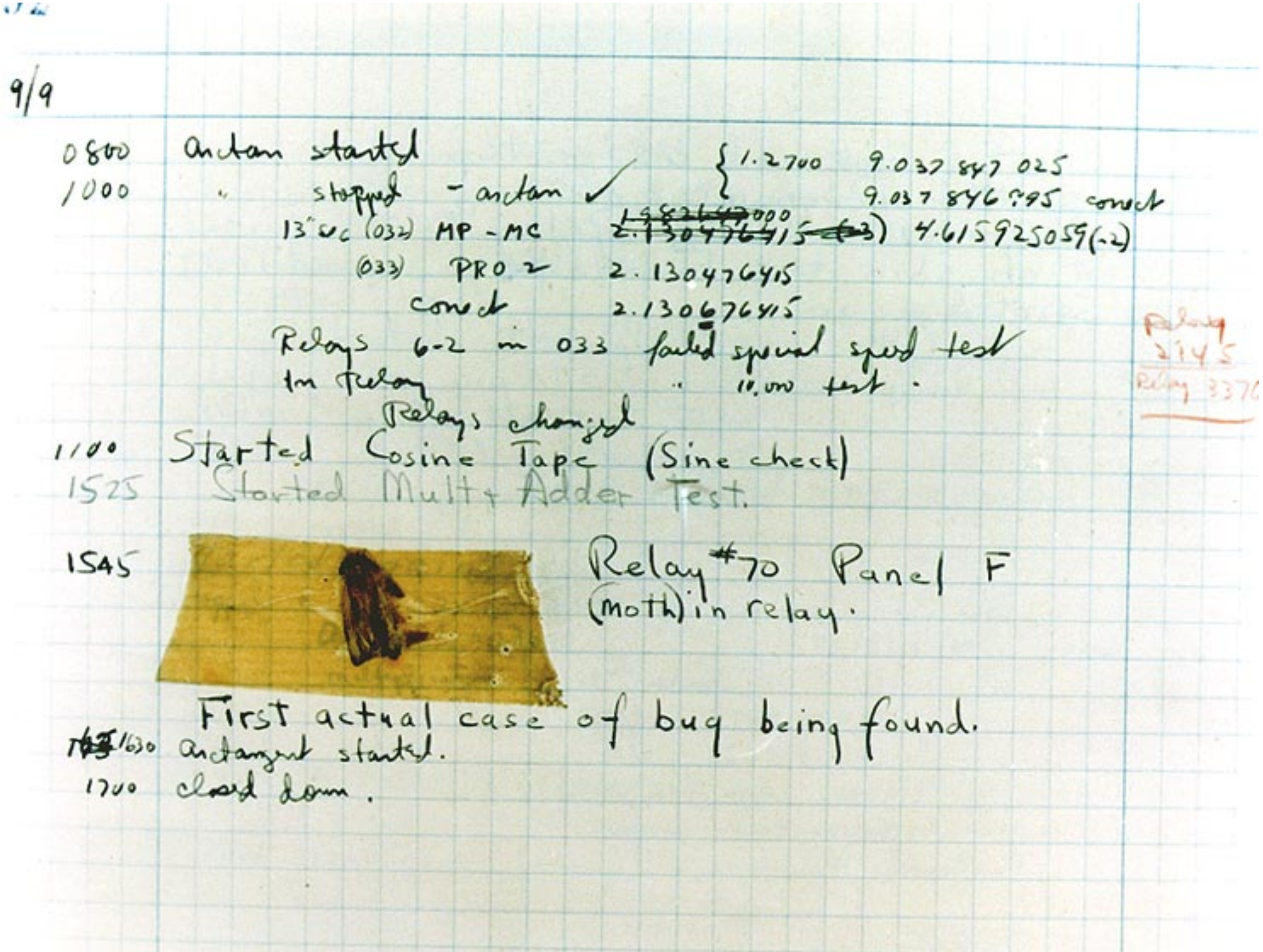
Control Flow, Debugging, and Testing

Dr.-Ing. Sebastian Proksch

University of Zurich, Department of Informatics

What is a bug?


9/9
0800 Anctan started
1000 stopped - anctan
1.2700 9.037 847 025
9.037 846 995 const
4.615925059(-2)
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
In relay
10,000 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.

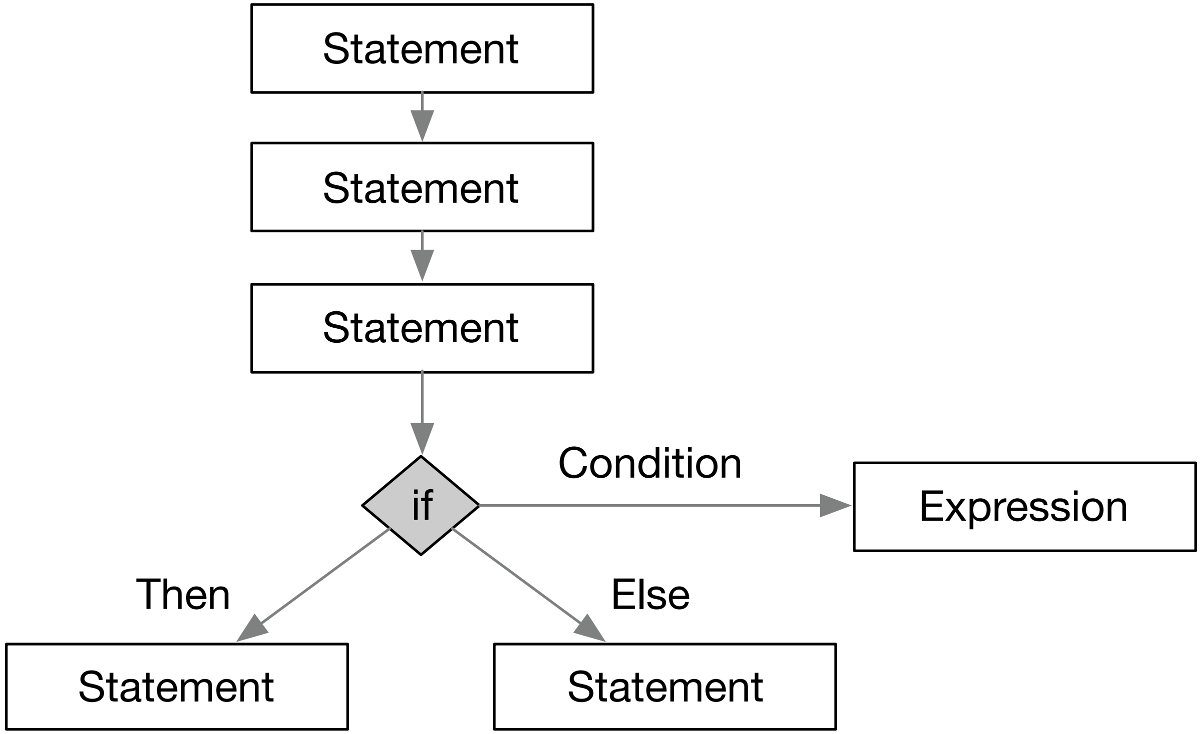
3.14
5^2
$2 \cdot 3$
$(1 - 2) + 3$
$101_2 = 5_{10}$
$\pi \approx 3.14$
$\sqrt{2}$
$0.999\ldots = 1$
$2 > -3$
$5(2 + 2)$
$1 + 2 \cdot 3$

Goals of Today

- Understand how a computer “remembers what to do”
- Learn how to trace a problem down to its cause
- Learn how to write repeatable and automatable tests

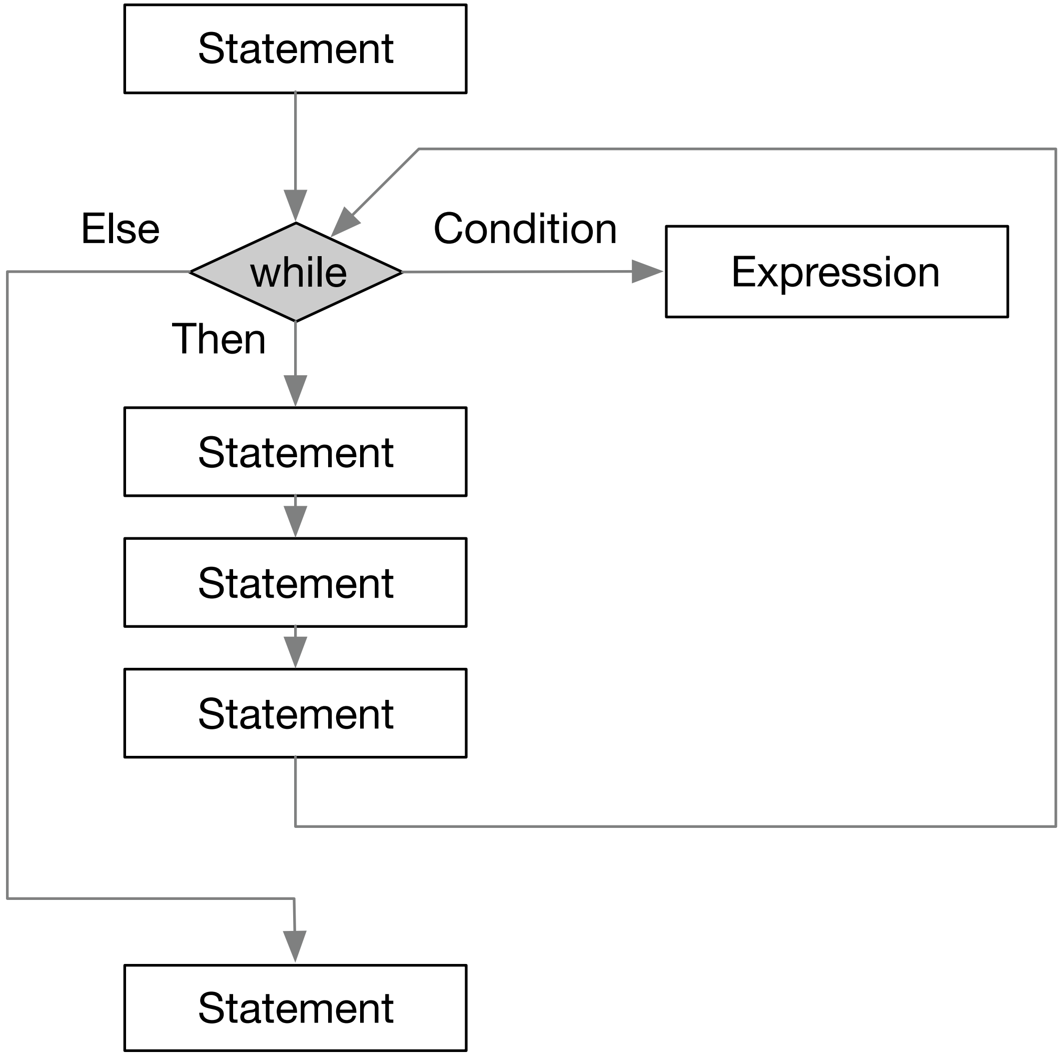
Control Flow

Statements and Branches



Name	Type	Value
x	int	…
y	int	…
z	float	…

Loops

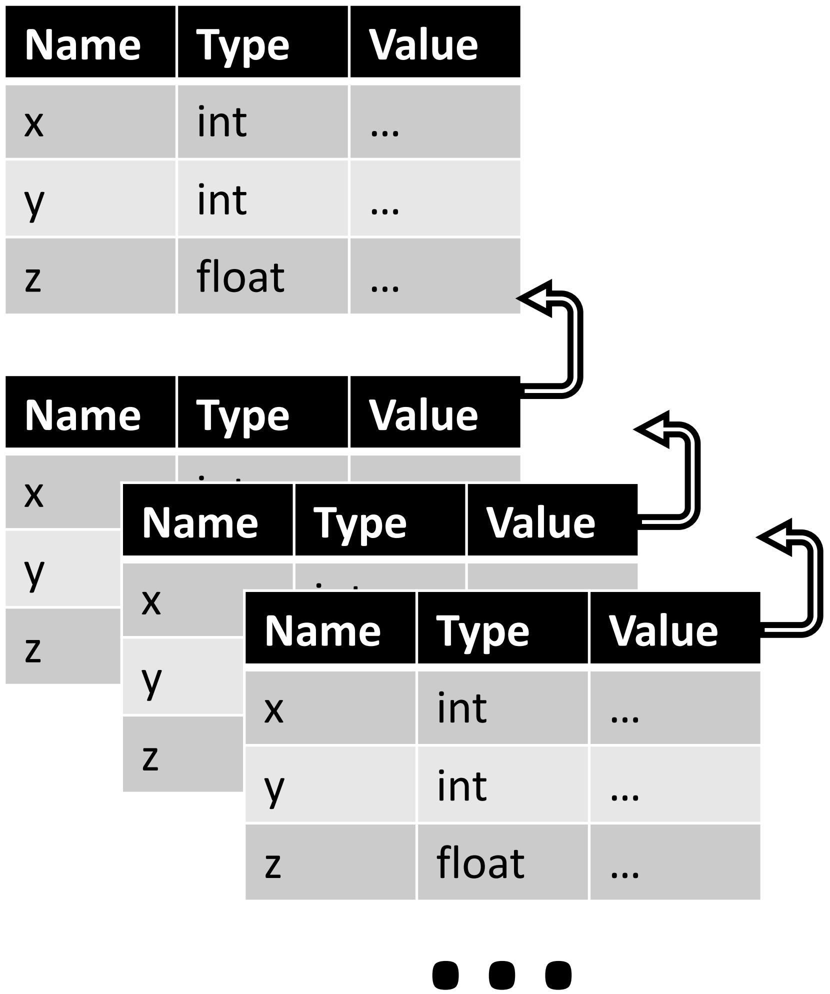


Name	Type	Value
x	int	…
y	int	…
z	float	…

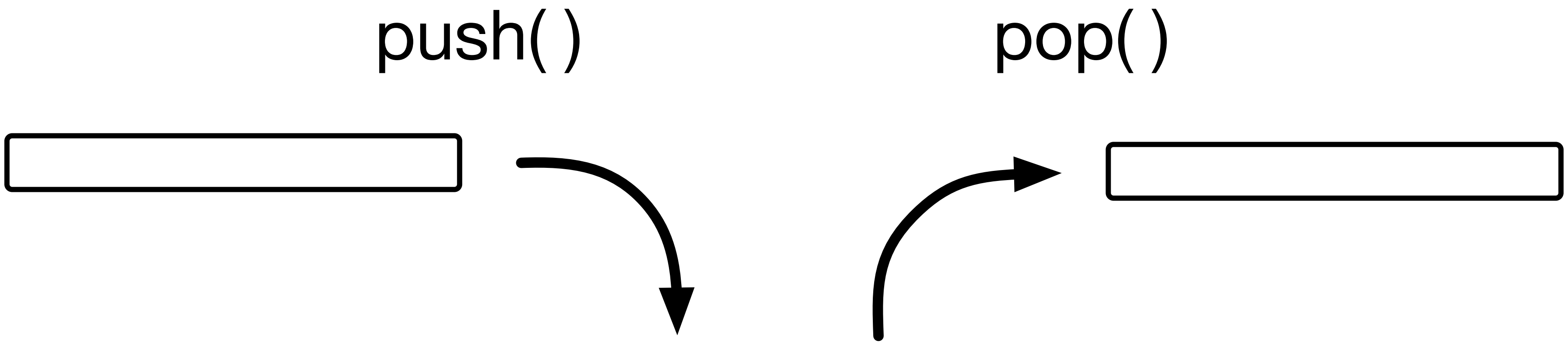
Functions/Methods

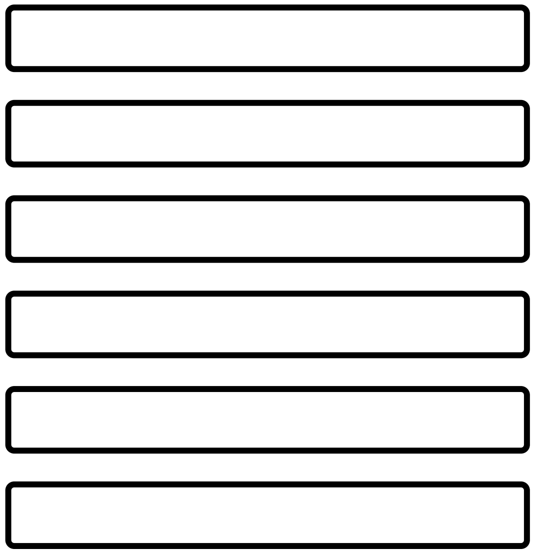
```
x = …
y = …
```

```
def f(x):
    y = …
    z = …

    f(x+1)
```



Excursion: What is a stack?

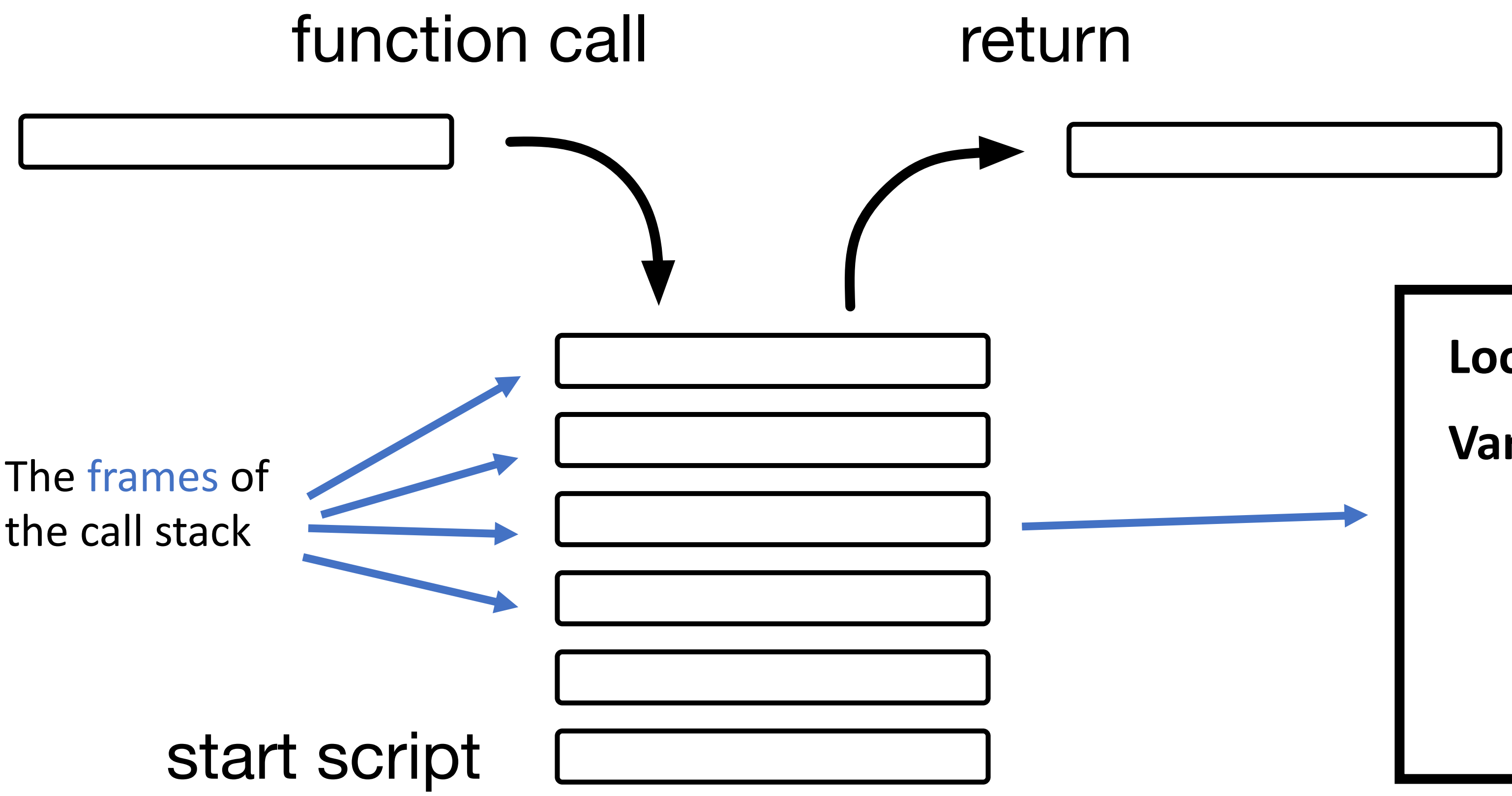




A stack is a data structure. It stores its data following a Last-In, First-Out principle (LIFO).

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

Let’s put together the pieces: the call stack!



Location: algorithm.py, line 10

Variables:

Name	Type	Value
x	int	…
y	int	…
z	float	…

Error Handling

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 not know that an error has occurred)
  - 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

```
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
  (out of scope for this lecture)

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")
```

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 included in a try block.

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

print("Live must go on.")
```

The except block specifies how the error is handled

In any case, execution continues after the try block

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))
    ...
```

A “Complete” Example of **try/except/…**

```
try:
    print("try1")
    fun_that_might_raise_ex()
    print("try2")
except ZeroDivisionError:
    print(”specific except")
except:
    print(“general except”)
else:
    print("else")
finally:
    print("finally")
```

Try the following…

… and handle specific exceptions.

… or fall back to general catch

If no exception was thrown…

In the end, always execute the following

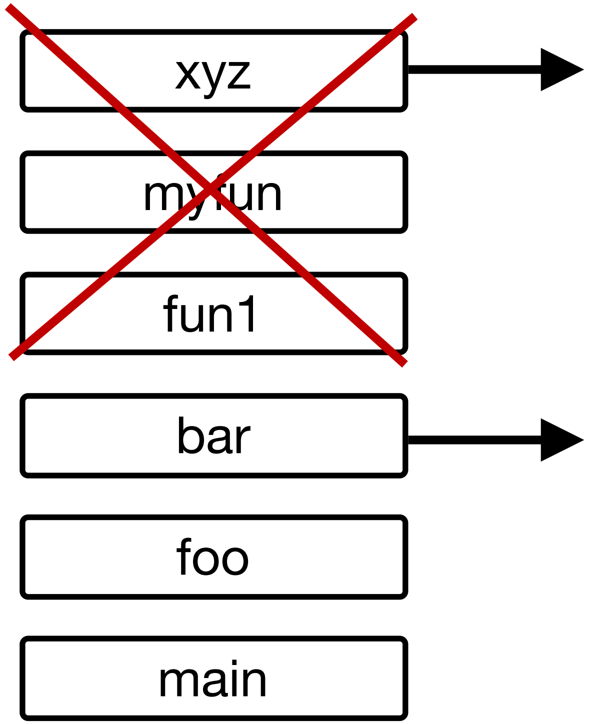
A Minimal Example of a try

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

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

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

What happens to the stack when an exception is raised?




```
def xyz(n):
    if n < 0:
        raise ValueError("meh!")
    ...
```

```
def bar():
    try:
        fun1("some text")
    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
e	tuple	("meh!",)

Goals of Today

- Understand how a computer "remembers what to do"
- Learn how to trace a problem down to its cause
- Learn how to write repeatable and automatable tests

Debugging

Breakpoints

```python
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

- Step Over
- Step Into
- Step Out
- Continue
- Abort

Step Over

"Execute the next statement"

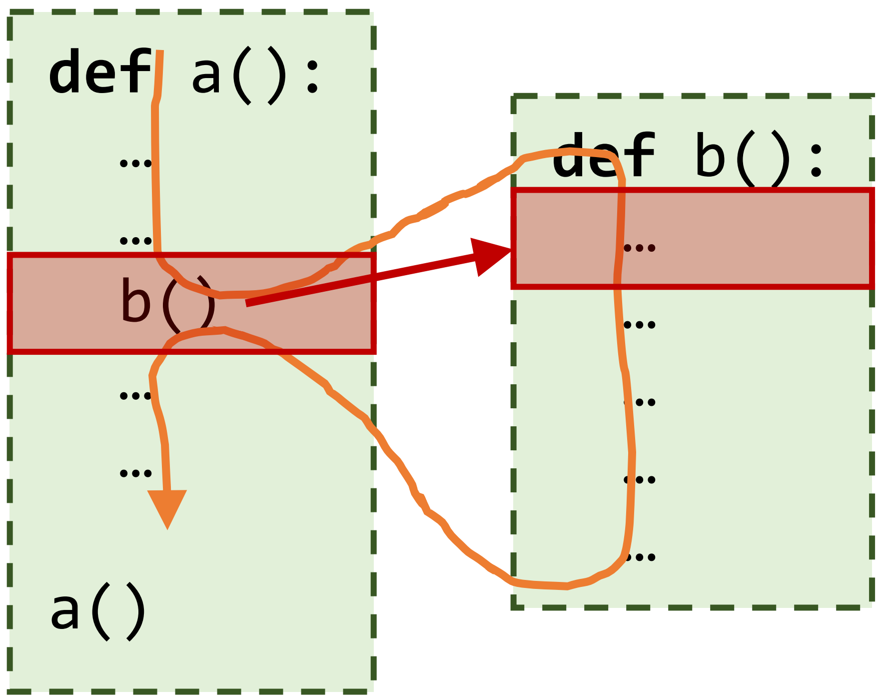
```
def a():
   x = 1
   y = 2
   z = x + y
   del(y)
   x += 1

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"

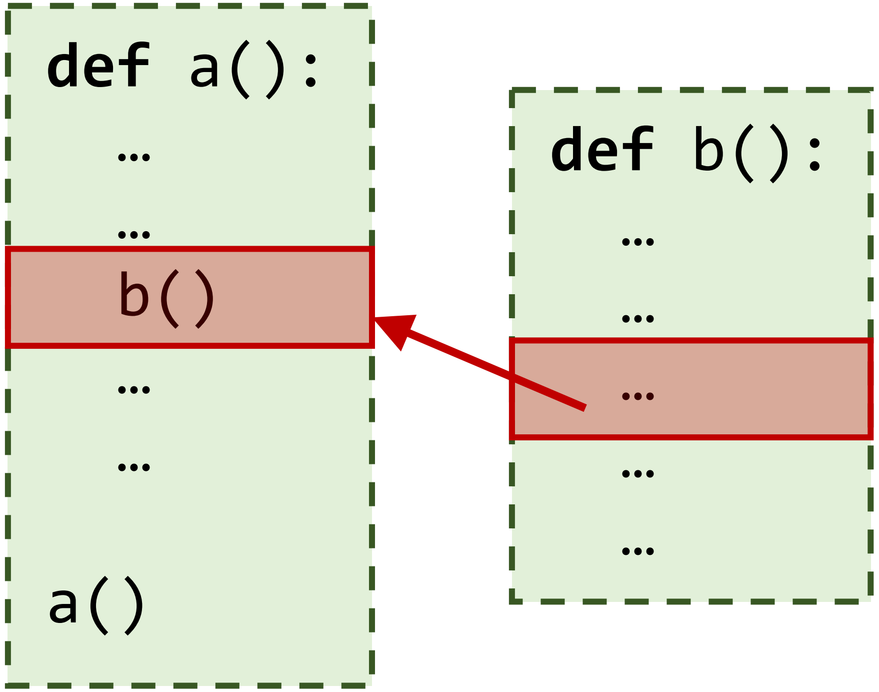


The current statement could be a call, which adds another stack frame. The debugger will *step into* this call and stop at the first statement in the new function.

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

Step Out

"Execute the rest of this function and go back to the caller"



A *step out* finishes the execution of the current function. The debugger will stop again when the current stack frame has been removed from the stack.

Continue

"Stop stepping and continue with regular program execution"

```
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.

Advanced Debugging

- 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)

Goals of Today

- Understand how a computer "remembers what to do"
- Learn how to trace a problem down to its cause
- Learn how to write repeatable and automatable tests

Testing

Assert That Your Assumptions Are Met

```python
def foo(x):
  assert x >= 0
  …

foo(-1) # AssertionError
```

In contrast to the more elaborated exceptions that we have used earlier, we can use `assert` to enforce specific preconditions in a program in a light-weight manner. All that `assert` does is checking whether the provided expression is `True` and raising an `AssertionError` if not.

Add Meaningful Error Messages

```
def foo(x):
  assert x >= 0, “x >= 0 is not true”
  …

foo(-1) # AssertionError: x >= 0 is not true
```

You Can Also Assert Outputs (Unit Tests)

```
def foo(x):
  …

assert foo(-1) == None
assert foo(0) == 1
assert foo(1) == 1
assert foo(2) == 2
…
```

Use `assert` in unit tests to validate the postconditions after various runs of a program.

This assertion style will be used in the exams to unambiguously define expected outputs.

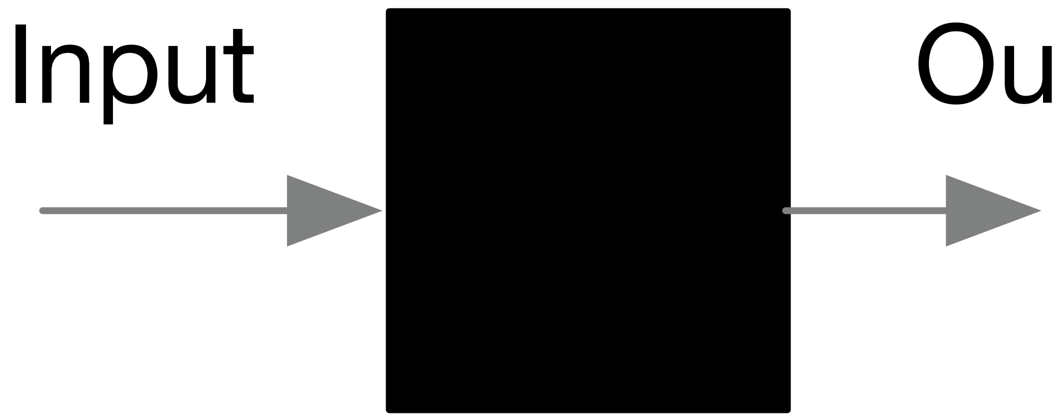
Why should I bother writing unit tests?

- Unit tests can be automated, it is cheap to test “everything at once”
- It is very easy to repeat a test
- Unit tests can serve as documentation
- A solid test suite provides a safety net and encourages refactorings

Refactoring: Changing the source code of a program to improve its quality, without changing its behavior.

How to find good assert cases?

Black-Box Testing



Interesting test cases are solely identified by studying the specification of a program, e.g., by testing relevant boundaries between value domains. This can be natural domains (e.g., positive and negative numbers) or problem-specific domains (e.g., age > 18).

White-box Testing

```
def foo(x, y):
  if x == 13:
    …
  elif x < y:
    …
  else
    for i in range(x, y):
      …
```

Interesting test cases are identified by studying the implementation of a program, e.g., by testing relevant conditions or case distinctions. White-box tests often try to maximize the count of tested lines in the program (test coverage)

Regression Testing

- Often, it is hard to understand and find a bug
- You can use debugging to explore erroneous executions
- Once understood, replicate the problem in a unit test
- The resulting regression test resembles this specific use case
- Regression tests prevent the same bug from being reintroduced

Test Generation

- Often, it is possible to write test code that automatically checks many different inputs for specific properties.

```
def square(x):
  …

for i in range(-100, 100):
  assert square(i) >= 0
```

Manual Testing

- Sometimes it is very hard to define automated test suites. For example, when network accesses, UI interactions, or system events are involved. Instead of testing individual parts with automated unit tests, it can be easier to manually test the whole system at once.
- It is important to execute this testing strategy in a structured fashion to make sure that important use cases are covered and that no corner cases are forgotten.
- It is crucial to formulate test plans that reflect which actions should be tested in which environment and to define the expected output.

Important Points to Define in a Test Plan

- What is the initial state of my application? How do I get there?
- Which exact steps do I need to perform for the test?
- What do I need to check to assert correctness?

Automated Unit Testing

Four-Phase Unit Testing

Each programmed unit should be tested. This should be done in isolation to avoid side effects introduced through mistakes in other units.

Such a unit test follows four phases:

- Setup (Prepare the environment)
- **Exercise** (Interact with the *system-under-test*, sut)
- **Verify** (assert that the output/state is correct)
- Teardown (Bring everything back to the original state)

Please note that exercise and verify are the crucial phases. Setup and teardown are optional and not needed, most of the time.

Poor Man's Testing "Framework"

```python
def square(x):
    return x * x

if __name__ == "__main__":
    assert square(0) == 0
    assert square(1) == 1
    ...
```

Filter by `__name__` to execute the asserts only when the file is executed directly.

You don't have to, though. `asserts` should be transparent, but not filtering means they are executed on every `import`.

This testing style bloats the source file, gets chaotic quickly, and is hard to debug, because different tests are tangled.

Excursion: Using Different Source-Code Files

mathutils.py

```
def square(x):
  return x * x
```

my_program.py

```
from mathutils import square

print (“3^2 = %d” % square(3))
```

We will talk more about how to use modules in Python in the upcoming lecture “Organizing Code: Modules and Version Control”

Excursion: Classes

```python
class C:

    def set_x(self, x):
        self.x = x

    def get_x(self):
        return self.x
```

```python
c1 = C()
c1.set_x(1)

c2 = C()
c2.set_x(2)

print(c1.get_x()) # 1
print(c2.get_x()) # 2
```

For now, think of classes as “bundles of coherent functions with a state”. We will learn more in the upcoming lectures about “Object-Oriented Programming”

Defining Unit Tests in **`unittest`**

(Python's built-in unit testing framework)

mathutils.py

```
def square(x):
  return x * x
```

A good unit test should only test one thing and, therefore, should have exactly one assert!

square_test.py

```
from unittest import TestCase
from mathutils import square

class square_test(TestCase):

   def test_zero(self):
      actual = square(0)
      expected = 0
      self.assertEqual(expected, actual)

   def test_something_else(self):
      ...
```

exercise

verify

Pick the right assert method

- Equality:
  - assertEqual
  - assertIs
- Relations:
  - assertGreaterThan
  - assert
- Exceptions:
  - assertRaises
- ...

Keep the test short and readable by using the right assertion method!

https://docs.python.org/3/library/unittest.html#assert-methods

Goals of Today

- ✓ Understand how a computer "remembers what to do"
- ✓ Learn how to trace a problem down to its cause
- ✓ Learn how to write repeatable and automatable tests

You should be able to answer the following:

- What is a call stack? A Frame?
- How is control flow represented in the Python interpreter?
- What are exceptions? How do you raise and handle them?
- How can I stop a program execution at at specific point?
- Which tools do I have to interact with a running program?
- When should I debug, when should I write unit tests?
- What is the dis-/advantage of unit testing, compared to debugging?
- What is the difference between white-box and black-box testing?
- What is the relation between unit tests and regression tests?
- How can I use “unittest” to write unit tests in Python?

Exercise

Exercise

- Take one (any!) function that has caused you headache recently
- Debug the function to understand what went wrong
- Write a test that reproduces an incorrect behavior
- Fix the bug to fix the test

A possible alternative: Write/debug/test a function `ls` that takes a string `str` and extracts the longest substring that only contains characters in alphabetical order. If alternatives exist, pick the first one. (e.g., `ls("xabcdagfe")`→`"abcd"`, `ls("xyzabc")`→`"xyz"`)