DM550/DM857
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM550/

http://imada.sdu.dk/~petersk/DM857/

Operator Precedence

- expressions are evaluated left-to-right
  - Example: 64 - 24 + 2 == 42
- BUT: like in mathematics, “*” binds more strongly than “+”
  - Example: 2 + 8 * 5 == 42
- parentheses have highest precedence: 64 - (24 + 2) == 38
- PEMDAS rule:
  - Parentheses “(<expr>)”
  - Exponentiation “**”
  - Multiplication “*” and Division “/”, "//", "%"
  - Addition “+” and Subtraction “-”

String Operations

- Addition “+” works on strings:
  - Example 1: `print("Hello w" + "orld!")`
  - Example 2: `print("4" + "2")`

- Multiplication “*” works on strings, if 2nd operands is integer:
  - Example: `print("Hej!" * 10)`

- Subtraction “-”, Division “/”, and Exponentiation “**” do NOT work on strings

Debugging Expressions

- most beginners struggle with common Syntax Errors:
  - check that all parentheses and quotes are closed
  - check that operators have two operands
  - sequential instruction should start on the same column or be separated by a semicolon “;”

- common Runtime Error due to misspelling variable names:
  - Example:

```
a = float(input()); b = float(input())
reslut = a**b+b**a
print(result)
```

Statements

- instructions in Python are called *statements*

- so far we know 2 different statements:
  - assignments "=": c = a**2+b**2
  - any expression is a statement

- as a grammar rule:

```
<stmt>    =>      <var> = <expr>        |
                  <expr>
```

Comments

- programs are not only written, they are also read

- document program to provide intuition:
  - Example 1: c = sqrt(a**2+b**2) # use Pythagoras
  - Example 2: x, y = y, x # swap x and y

- all characters after the comment symbol "#" are ignored
  - Example: x = 23 #+19

    results in x referring to the value 23

Many Ways to Python Development

- browser-based development:
  - https://trinket.io/features/python3
- browser-based visualization:
  - http://www.pythontutor.com/visualize.html

- standard IDE for Python NOT recommend (IDLE)
- other Python(-enabled) IDEs: https://wiki.python.org/moin/IntegratedDevelopmentEnvironments
- Thonny (incl. visualization): http://thonny.org/
- Pyzo + Miniconda/Anaconda: http://pyzo.org/
- or for something else: https://ipython.org/

- hardcore: use command line and a text editor :)

Code Café

- manned Code Cafe for students
- first time Wednesday, September 6
- last time Wednesday, December 20
- closed in Week 42 (efterårsferie)

- Mondays, 15.00 – 17.00, Nicky Cordua Mattsson
- Wednesdays, 15.00 – 17.00, Troels Risum Vigsøe Frimer

- Nicky and Troels can help with any coding related issues
- issues have to be related to some IMADA course (fx this one)

CALLING FUNCTIONS

Calling Functions

- so far we have seen four different *function calls*:
  - input(): reads a value from the keyboard
  - sqrt(x): computes the square root of x
  - type(x): returns the type of the value of x
  - print(x): prints argument and returns None

- in general, a function call is also an expression:
  - <expr> => … | <function>(<arg$_1$>, …, <arg$_n$>)
  - Example 1:
    ```
    x = input()
    print(type(x))
    ```
  - Example 2:
    ```
    from math import log
    print(log(4398046511104, 2))
    ```

Importing Modules

- we imported the sqrt function from the math module:

```
from math import sqrt
```

- alternatively, we can import the whole module:

```
import math
```

- using the built-in function "dir(x)" we see math's functions:

```
acos   ceil     expm1     gamma    ldexp   pow      trunc
acosh  cos      fabs      gcd      lgamma  radians
asin   cosh     factorial hypot    log     sin
asinh  degrees  floor     isclose  log10   sinh
atan   erf      fmod      isfinite log1p   sqrt
atan2  erfc     frexp     isinf    log2    tan
atanh  exp      fsum      isnan    modf    tanh
```

- access using "math.<function>":   c = math.sqrt(a**2+b**2)

The Math Module

- contains 25 functions (trigonometric, logarithmic, …):
  - Example: x = input()
    print(math.sin(x)**2+math.cos(x)**2)
- contains 2 constants (math.e and math.pi):
  - Example: print(math.sin(math.pi / 2))
- contains 3 meta data (__doc__, __file__, __name__):
  - print(math.__doc__)
  - print(math.frexp.__doc__)
  - print(type.__doc__)

Type Conversion Functions

- Python has pre-defined functions for converting values

- int(x): converts x into an integer
  - Example 1: int("1234") == int(1234.9999)
  - Example 2: int(-3.999) == -3

- float(x): converts x into a float
  - Example 1: float(42) == float("42")
  - Example 2: float("Hej!") results in Runtime Error

- str(x): converts x into a string
  - Example 1: str(23+19) == "42"
  - Example 2: str(type(42)) == "<type 'int'>"

DEFINING FUNCTIONS

Function Definitions

- functions are defined using the following grammar rule:

  <func.def> => def <function>(<arg$_1$>, …, <arg$_n$>):

  <instr$_1$>; …; <instr$_k$>

- can be used to reuse code:
  - Example:

```
def pythagoras():
    c = math.sqrt(a**2+b**2)
    print("Result:", c)
a = 3; b = 4; pythagoras()
a = 7; b = 15; pythagoras()
```

- functions are values: `type(pythagoras)`

Functions Calling Functions

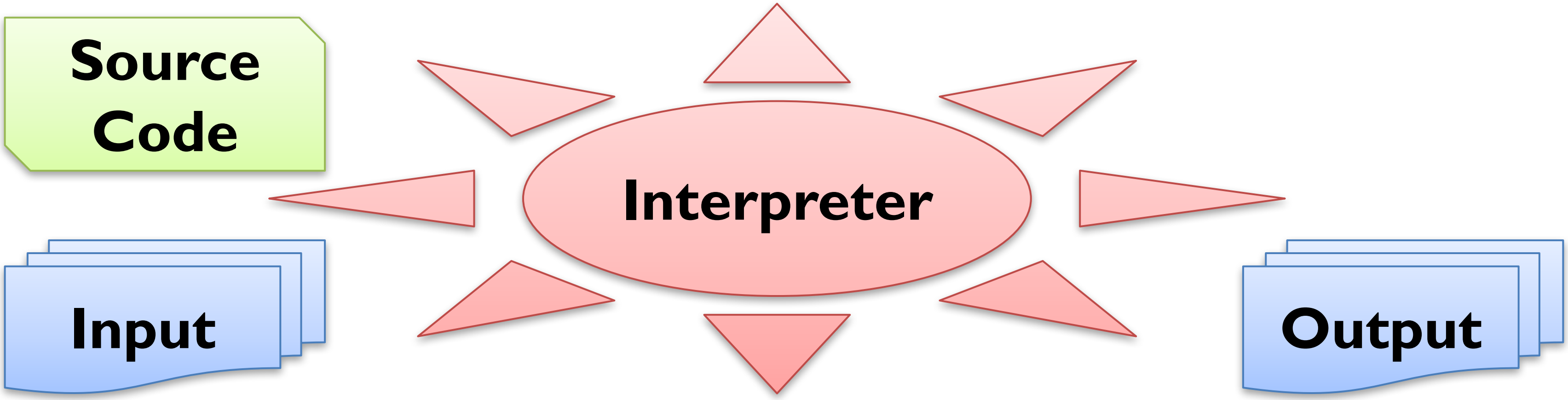
- functions can call other functions

- Example:

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

Executing Programs (Revisited)

- Program stored in a file (*source code* file)
- Instructions in this file executed top-to-bottom
- Interpreter executes each instruction



Functions Calling Functions

- functions can call other functions

- Example:

create new function variable “white”

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

create new function variable “black”

Functions Calling Functions

- functions can call other functions
- Example:

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

create new function variable “all”

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

call function “all”

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

call function “white”

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

print
" # # # # # # # #"

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

call function “black”

Functions Calling Functions

- functions can call other functions
- Example:

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

print
"# # # # # # # # "

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

call function “white”

Functions Calling Functions

- functions can call other functions

- Example:

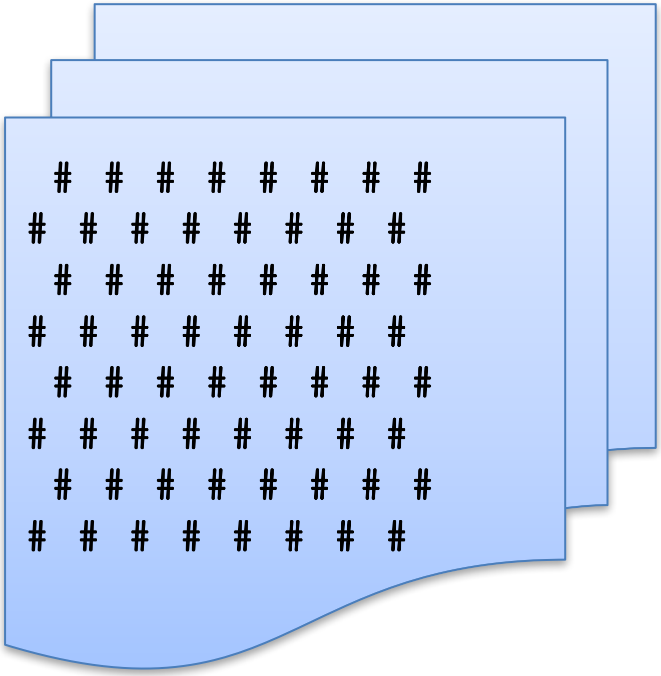
```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

print
" # # # # # # # #"

Functions Calling Functions

- functions can call other functions

- Example:



```
 # # # # # # # #
# # # # # # # # 
 # # # # # # # #
# # # # # # # # 
 # # # # # # # #
# # # # # # # # 
 # # # # # # # #
# # # # # # # # 
```

```
def white():
    print(" #" * 8)
def black():
    print("# " * 8)
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras():
    c = math.sqrt(a**2+b**2)
    print("Result:", c)
a = 3; b = 4; pythagoras()
a = 7; b = 15; pythagoras()
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print("Result:", c)
a = 3; b = 4; pythagoras(a, b)
a = 7; b = 15; pythagoras(a, b)
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print("Result:", c)
pythagoras(3, 4)
pythagoras(7, 15)
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print("Result:", c)
pythagoras(3, 4)
pythagoras(2**3-1, 2**4-1)
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print("Result:", c)
pythagoras(3, 4)
x = 2**3-1; y = 2**4-1
pythagoras(x, y)
```

Variables are Local

- parameters and variables are local
- local = only available in the function defining them
- Example: in module math:

```
def sqrt(x):
    …
```

x local to math.sqrt

a local to pythagoras

b local to pythagoras

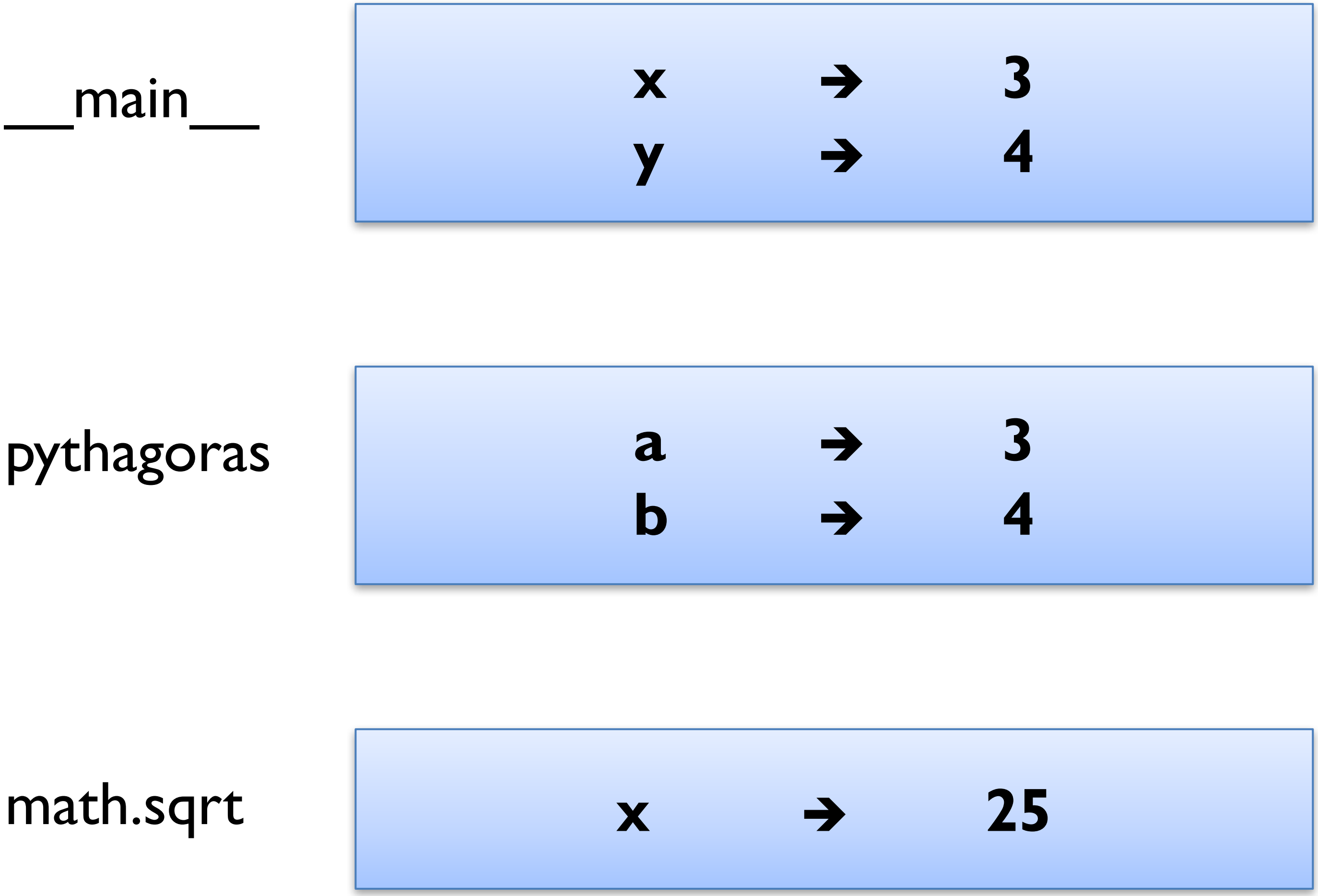
c local to pythagoras

x,y local to __main__

in our program:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print("Result:", c)
x = 3;  y =4;  pythagoras(x, y)
```

Stack Diagrams



__main__

x → 3

y → 4

pythagoras

a → 3

b → 4

math.sqrt

x → 25

Tracebacks

- stack structure printed on runtime error
- Example:

```
def broken(x):
    print(x / 0)
def caller(a, b):
    broken(a**b)
caller(2,5)
```

```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    caller(2,5)
  File "test.py", line 4, in caller
    broken(a**b)
  File "test.py", line 2, in broken
    print(x/0)
ZeroDivisionError: integer division or modulo by zero
```

Return Values

- we have seen functions that return values:
  - math.sqrt(x) returns the square root of x
  - math.log(x, base) returns the logarithm of x w.r.t. base
- What is the return value of our function pythagoras(a, b)?
- special value None returned, if no return value given (*void*)
- declare return value using return statement: return <expr>
- Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    return c
print(pythagoras(3, 4))
```

Motivation for Functions

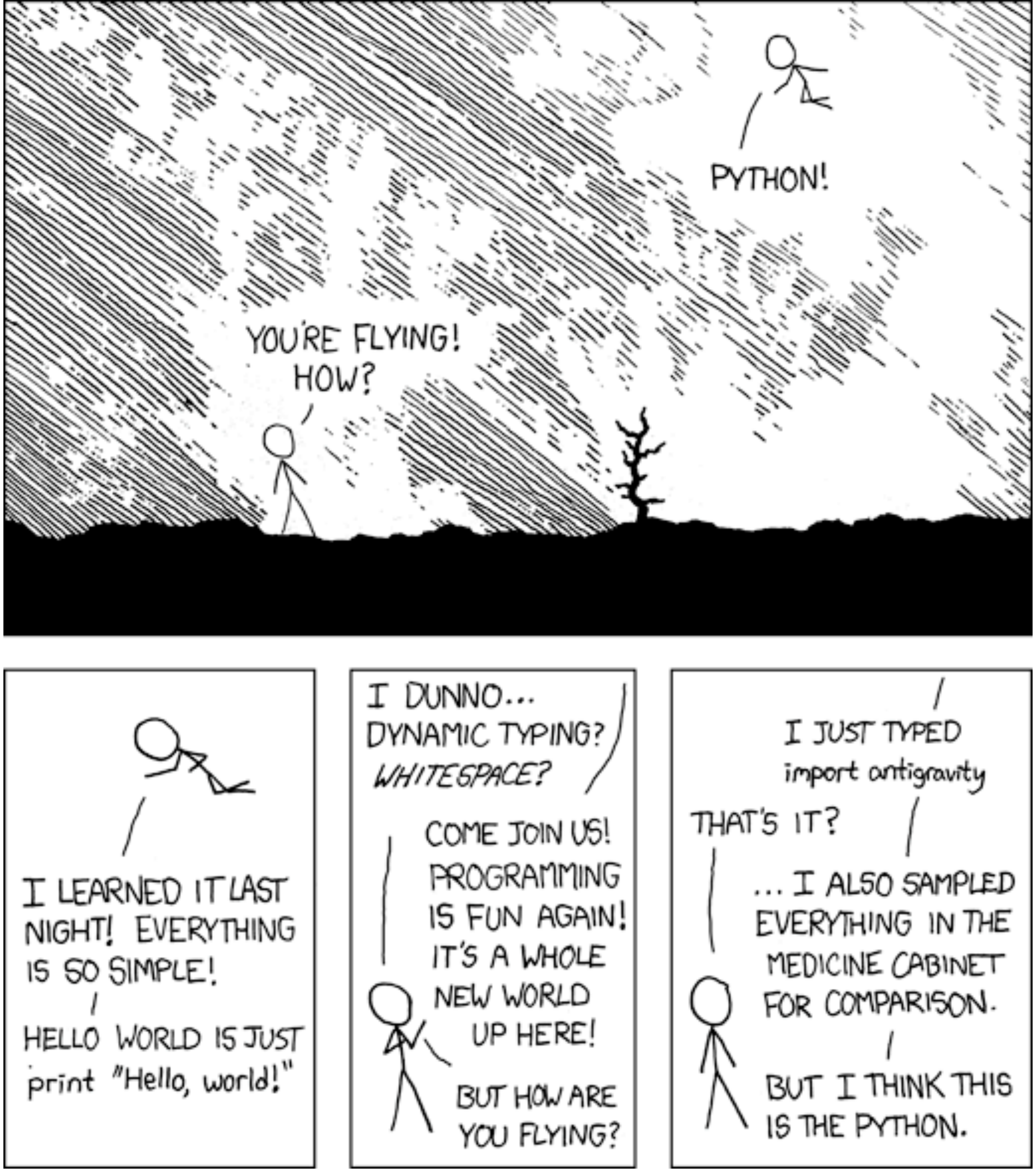
- functions give names to blocks of code
  - easier to read
  - easier to debug
- avoid repetition
  - easier to make changes
- functions can be debugged separately
  - easier to test
  - easier to find errors
- functions can be reused (for other programs)
  - easier to write new programs

Debugging Function Definitions

- make sure you are using latest files (save, then run python -i)

- biggest problem for beginners is *indentation*
  - all lines on the same level must have the same indentation
  - mixing spaces and tabs is very dangerous
  - try to use only spaces – a good editor helps!

- do not forget to use ":" at end of first line
- indent body of function definition by e.g. 4 spaces


YOU'RE FLYING! HOW?
PYTHON!
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

TURTLE GRAPHICS & INTERFACE DESIGN

Turtle Module

- available in most Python distributions
- easy to use (although requires some faith at the moment)
  - can be imported using import turtle
  - t = turtle.Turtle() creates new turtle t
  - turtle.mainloop() can be used at the end of the program

- two basic commands to the turtle
  - t.fd(x) advances turtle t by x units
  - t.lt(x) turns turtle t x degrees to the left

Simple Repetition

- drawing a square by e.g. 4x drawing a line and turning left

```
t.fd(100); t.lt(90); t.fd(100); t.lt(90);
t.fd(100); t.lt(90);  t.fd(100); t.lt(90)
```

- simple repetition using for-loop

```
for <var> in range(<expr>):
    <instr_1>;  <instr_2>
```

- Example:

```
for i in range(4):
    print(i)
```

Simple Repetition

- drawing a square by e.g. 4x drawing a line and turning left

```
t.fd(100); t.lt(90); t.fd(100); t.lt(90);
t.fd(100); t.lt(90);  t.fd(100); t.lt(90)
```

- simple repetition using for-loop

```
for <var> in range(<expr>):
    <instr_1>;  <instr_2>
```

- Example:

```
for i in range(4):
    t.fd(100)
    t.lt(90)
```

Encapsulation

- **Idea:** wrap up a block of code in a function
  - documents use of this block of code
  - allows reuse of code by using parameters

- Example:

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)
square(t)
u = turtle.Turtle(); u.rt(90); u.fd(10); u.lt(90);
square(u)
```

Generalization

- square(t) can be reused, but size of square is fixed
- **Idea:** generalize function by adding parameters
  - more flexible functionality
  - more possibilities for reuse
- Example 1:

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)
square(t, 100)
square(t, 50)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def square(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, length):
    for i in range(4):
        t.fd(length)
        t.lt(90)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    for i in range(n):
        t.fd(length)
        t.lt(90)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    for i in range(n):
        t.fd(length)
        t.lt(360/n)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
polygon(t, 4, 100)
polygon(t, 6, 50)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
polygon(t, n=4, length=100)
polygon(t, n=6, length=50)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        t.fd(length)
        t.lt(angle)


square(t, 100)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        t.fd(length)
        t.lt(angle)
def square(t, length):
    polygon(t, 4, length)
square(t, 100)
```

Interface Design

- **Idea:** interface = parameters + semantics + return value
- should be general (= easy to reuse)
- but as simple as possible (= easy to read and debug)

- Example:

```
def circle(t, r):
    circumference = 2*math.pi*r
    n = 10
    length = circumference / n
    polygon(t, n, length)
circle(t, 10)
circle(t, 100)
```

Interface Design

- **Idea:** interface = parameters + semantics + return value
- should be general (= easy to reuse)
- but as simple as possible (= easy to read and debug)

- Example:

```
def circle(t, r, n):
    circumference = 2*math.pi*r
#   n = 10
    length = circumference / n
    polygon(t, n, length)
circle(t, 10, 10)
circle(t, 100, 40)
```

Interface Design

- **Idea:** interface = parameters + semantics + return value
- should be general (= easy to reuse)
- but as simple as possible (= easy to read and debug)

- Example:

```
def circle(t, r):
    circumference = 2*math.pi*r
    n = int(circumference // 3) + 1
    length = circumference / n
    polygon(t, n, length)
circle(t, 10)
circle(t, 100)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n


    for i in range(n):
        t.fd(step_length)
        t.lt(step_angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n

def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
def polygon(t, n, length):
    angle = 360/n
    polyline(t, n, length, angle):
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
def circle(t, r):
    arc(t, r, 360)
```

Simple Iterative Development

- first structured approach to develop programs:
  1. write small program without functions
  2. encapsulate code in functions
  3. generalize functions (by adding parameters)
  4. repeat steps 1–3 until functions work
  5. refactor program (e.g. by finding similar code)

- copy & paste helpful
  - reduces amount of typing
  - no need to debug same code twice

Debugging Interfaces

- interfaces simplify testing and debugging

1. test if pre-conditions are given:
   - do the arguments have the right type?
   - are the values of the arguments ok?

1. test if the post-conditions are given:
   - does the return value have the right type?
   - is the return value computed correctly?

1. debug function, if pre- or post-conditions violated

GETTING YOUR HANDS DIRTY

Programming by Checklist ☺

1. Do you have an editor installed? (Preferably with syntax highlighting!)
2. Do you have a Python distribution installed?
3. Test the following programs:

   a.
   ```
   print("My name is Slartibartfast!")
   ```

   b.
   ```
   import turtle; t = turtle.Turtle()
   for i in range(4):
       t.fd(10); t.lt(270)
   ```

   c.
   ```
   import turtle; t = turtle.Turtle()
   list(map(lambda x:(t.fd(5*x),t.lt(120)),range(20)))
   ```

4. Extend to a meaningful program and save it !