DM536 / DM550 Part I
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM536/

CALLING FUNCTIONS

Calling Functions

- so far we have seen three 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

- 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(439804651111104, 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	cos	floor	log	sin
asin	cosh	fmod	log10	sinh
atan	degrees	frexp	modf	sqrt
atan2	exp	hypot	pow	tan
ceil	fabs	ldexp	radians	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

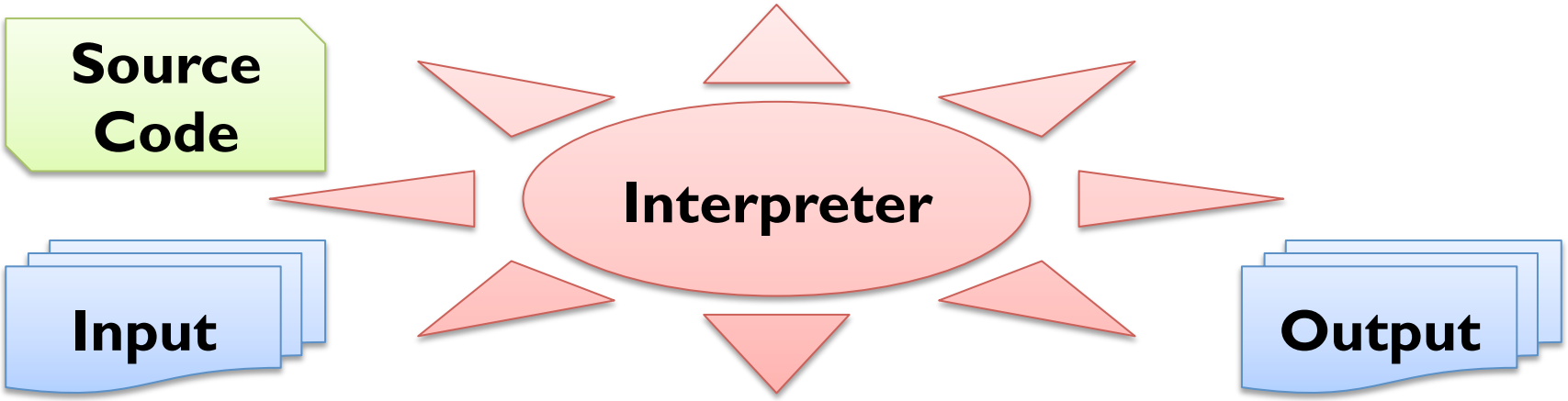
- functions can call other functions

- 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:

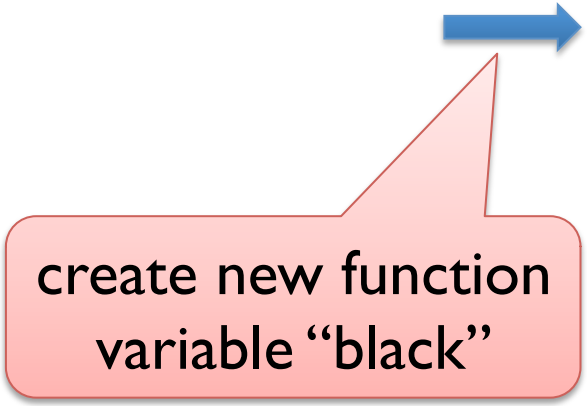
create new function variable "white"

```
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()
```

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()
```

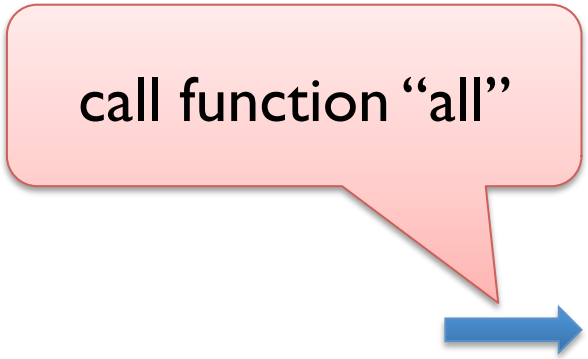
create new function variable “all”

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()
```

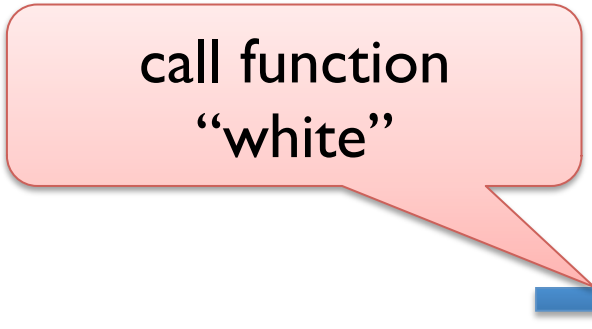


Functions Calling Functions

- functions can call other functions

- Example:

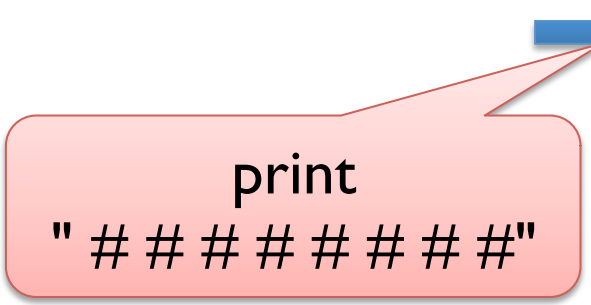
```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```



Functions Calling Functions

- functions can call other functions

- Example:



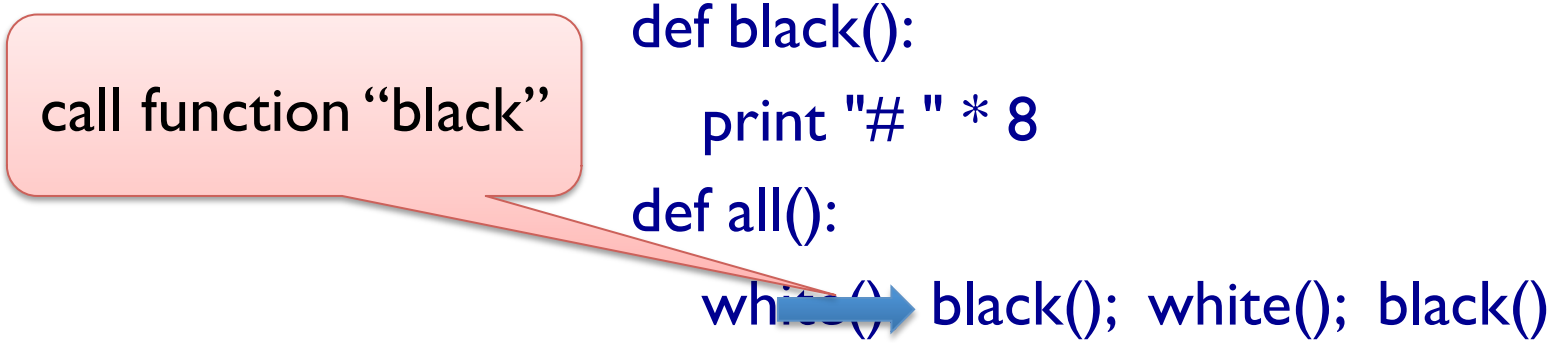

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

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()
```



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
"# # # # # # # # "

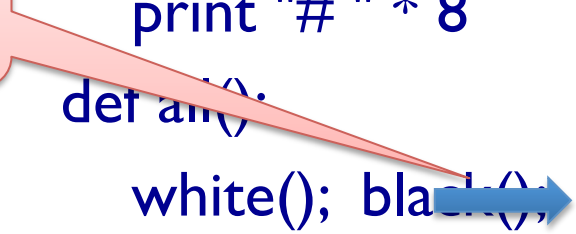
Functions Calling Functions

- 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()
```

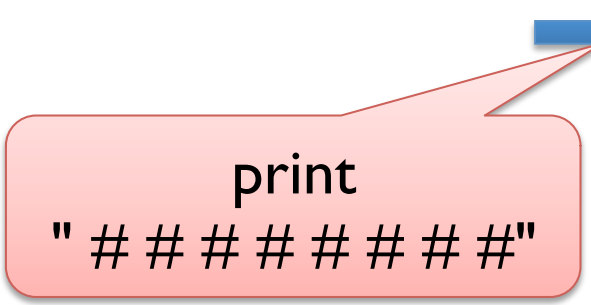
call function "white"



Functions Calling Functions

- functions can call other functions

- Example:

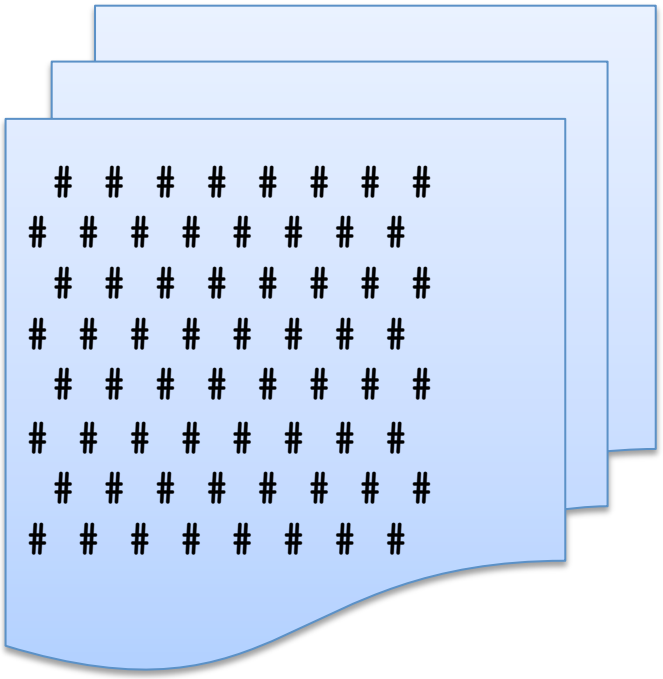



```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

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):
    …
```

in our program:

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

x local to math.sqrt

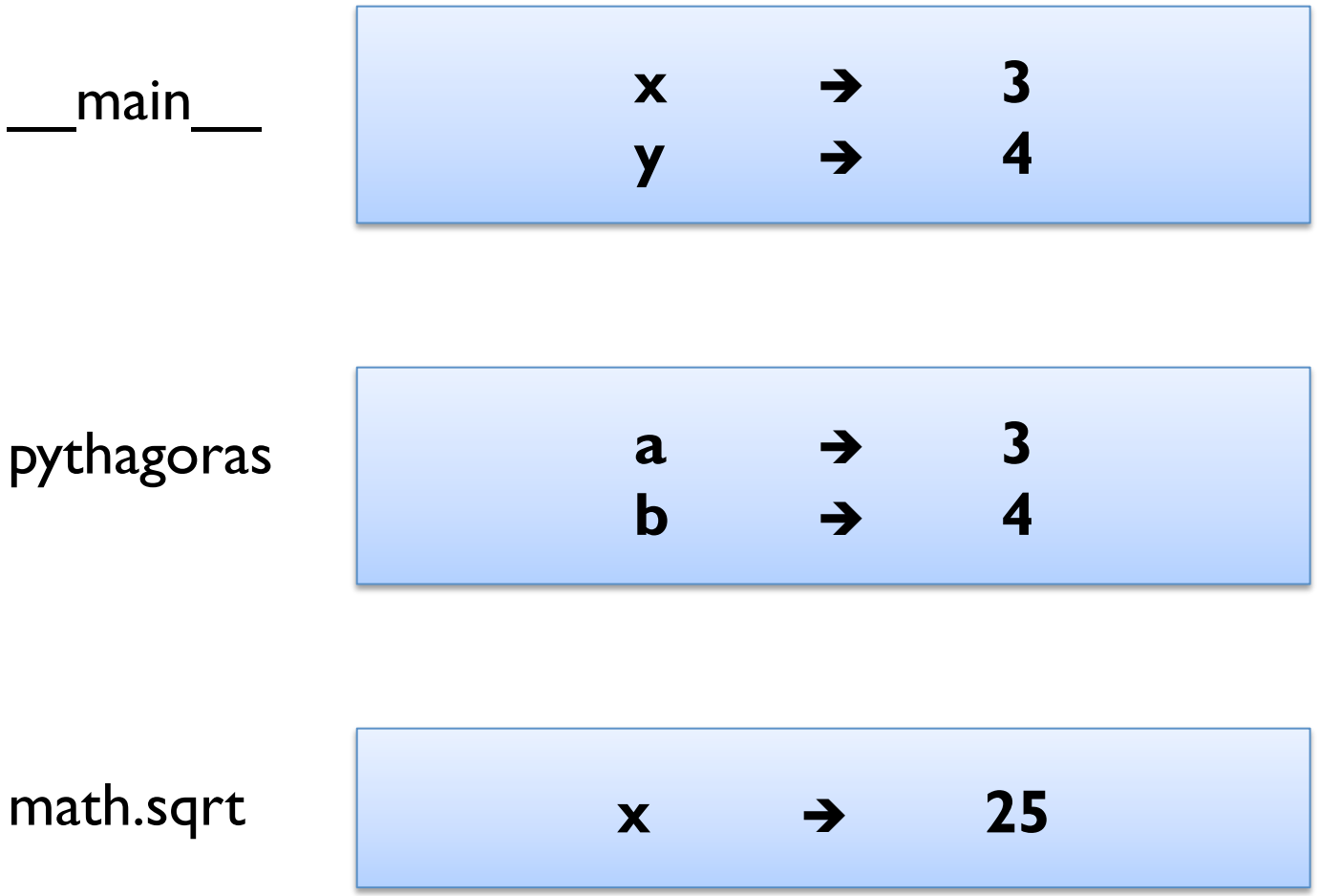
a local to pythagoras

b local to pythagoras

c local to pythagoras

x,y local to __main__

Stack Diagrams



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

TURTLE WORLD & INTERFACE DESIGN

Turtle World

- available from
  - http://www.greenteapress.com/thinkpython/swampy/install.html
- basic elements of the library
  - can be imported using from swampy.TurtleWorld import *
  - w = TurtleWorld() creates new world w
  - t = Turtle() creates new turtle t
  - wait_for_user() can be used at the end of the program

Simple Repetition

- two basic commands to the turtle
  - fd(t, 100) advances turtle t by 100
  - lt(t) turns turtle t 90 degrees to the left
- drawing a square requires 4x drawing a line and turning left
  - fd(t,100); lt(t); fd(t,100); lt(t); fd(t,100); lt(t); fd(t,100); lt(t)
- simple repetition using for-loop for <var> in range(<expr>):
  <instr$_1$>; <instr$_2$>
- Example:

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

Simple Repetition

- two basic commands to the turtle
  - fd(t, 100) advances turtle t by 100
  - lt(t) turns turtle t 90 degrees to the left
- drawing a square requires 4x drawing a line and turning left
  - fd(t,100); lt(t); fd(t,100); lt(t); fd(t,100); lt(t); fd(t,100); lt(t)
- simple repetition using for-loop `for <var> in range(<expr>):`
  `<instr1>; <instr2>`
- Example:

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

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):
        fd(t, 100)
        lt(t)
square(t)
u = Turtle(); rt(u); fd(u,10); lt(u);
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):
        fd(t, length)
        lt(t)
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):
        fd(t, length)
        lt(t)
```

Generalization

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

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

Generalization

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

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

Generalization

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

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

Refactoring

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

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

Refactoring

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

```
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, 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?
2. test if the post-conditions are given:
   - does the return value have the right type?
   - is the return value computed correctly?
3. debug function, if pre- or post-conditions violated