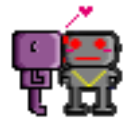
Introduction to Programming CS101

Spring 2012

Lecture #1



Welcome to CS101!

Welcome to Introduction to Programming!

CS101:

- 12 sections
- 6 lecturers
- 6 lead TAs
- 36 helper TAs

Each section has one 3-hour lab per week. This is the most important part of the course!

Two sections together have a lecture once a week:

- Mon. Sections A, B by Y.J. Lee; C, D by Moonzoo Kim;
- Wed. Sections E, F by Yu-Wing Tai; G, H by Ho-Jin Choi;
- Fri. Sections I, J by Taisook Han; K, L by Dong-Soo Han.

You must regularly check the course announcements on http://cs101.kaist.ac.kr

Grading

Practice points:

- 100 points for lecture attendance;
- 100 points for lab work;
- 200 points for homework.

Students need to collect at least 320 practice points.
Definitely NO BONUS Homework!

Theory points:

- 100 points for midterm exam;
- 100 points for final exam.

The grade is determined by the theory points only.
The practice points over 320 are ONLY qualification for grading.

Credit Transfer

Only by pass exam

- No credit transfer possible from other universities

Registration

- 2012.2.6. - 2.8

Pass Exam

- Time: 2012.2.13.(Monday) 8pm-11pm
- Location: Creative Learning Building Rm #307
- Language: Python
- Pass if above B-

Section Change/New Registration

Freshmen entering KAIST in 2012

- No section change is allowed.

New Registration

- Through online: KAIPA (http://kaipa.kaist.ac.kr)
- First Come First Serve
- Class size is limited to 45 students.
- 2012.2.6. - 2.17

Attendance

No attendance check in the first lecture.

When you come to the 2nd lecture

- Pick a seat and it will be your seat for the rest of the semester

Honesty policy

Cheating is strongly forbidden.

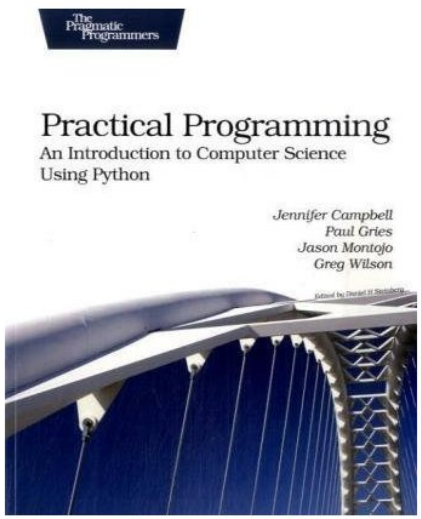
Cheating on homework or exams will give an F.

Course material

All course related materials will be made available on the course website.

- Lecture slides and example code
- Lecture notes on robot programming and photo processing
- Lab material
- `http://cs101.kaist.ac.kr/`

Main reference



Practical Programming: An Introduction to Computer Science Using Python by Jennifer Campbell, Paul Gries, Jason Montojo, and Greg Wilson. Pragmatic Programmers, 2009.

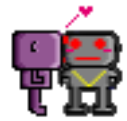
Python

This course is about programming and computational thinking, not about learning a programming language.

Python is a programming language that is easy to learn and very powerful.

- Used in many universities for introductory courses.
- Main language used for web programming at Google.
- Widely used in scientific computation, for instance at NASA, by mathematicians and physicists.
- Available on embedded platforms, for instance Nokia mobile phones.
- Large portions of games (such as Civilization IV) are written in Python.

Once you learnt programming in one language, it is relatively easy to learn another language, such as C++ or Java.



Why are you here?

Every scientist and engineer must know some programming. It is part of basic education, like calculus, linear algebra, introductory physics and chemistry, or English.

Alan Perlis 1961

Computer science is not computer programming. We teach programming to teach computational thinking:

- Solving problems (with a computer).
- Thinking on multiple levels of abstraction.
  Decompose a problem into smaller problems.
- A way of human thinking (not "thinking like a computer")
- Thinking about recipes (algorithms).
  30 years ago the solution to a problem in science or engineering was usually a formula. Today it is usually an algorithm (DNA, proteins, chemical reactions, factory planning, logistics).

What is a program?

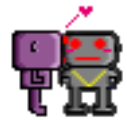
A program is a sequence of instructions that solves some problem (or achieves some effect).

For instance: Call your friend on the phone and give her instructions to find your favorite café. Or explain how to bake a cake.

Instructions are operations that the computer can already perform.

But we can define new instructions and raise the level of abstraction!

A program implements an algorithm (a recipe for solving a problem).



What is debugging?

A bug is a mistake in a program. Debugging means to find the mistake and to fix it.

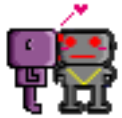
Computer programs are very complex systems. Debugging is similar to an experimental science: You experiment, form hypotheses, and verify them by modifying your program.

Kinds of errors:

- Syntax error. Python cannot understand your program, and refuses to execute it.
- Runtime error. When executing your program (at runtime), your program suddenly terminates with an error message.
- Semantic error. Your program runs without error messages, but does not do what it is supposed to do.

Why is programming useful?

- 20 years ago, electrical engineering students learnt about circuits. Today they learn about embedded systems.
- You can build a radio in software.
- Industrial engineers program industrial robots. Moreover, today's industrial engineers work on logistics—problems that can only be solved by computer.
- Modern automobiles contain thousands of lines of code, and would not run without microprocessors.
- Mathematicians gain insight and intuition by experimenting with mathematical structures, even for discrete objects such as groups and graphs.
- Experimental data often needs to be reformatted to be analyzed or reused in different software. Python is fantastic for this purpose.
- Experimental data sets are nowadays too large to be handled manually.



Why learn programming?

- If you can only use software that someone else made for you, you limit your ability to achieve what you want.
- For instance, digital media is manipulated by software. If you can only use Photoshop, you limit your ability to express yourself.
- Programming gives you freedom.

Why is programming fun?

Programming is a creative process.

A single person can actually build a software system of the complexity of the space shuttle hardware. Nothing similar is true in any other discipline.

There is a large and active open-source community: people who write software in their free time for fun, and distribute it for free on the internet. For virtually any application there is code available that you can download and modify freely.

Let's get started

But now let me show you some Python code...

- interactive Python
- Python programs (scripts)
- comments
- your own instructions: functions
- keywords

Adding new functions

A function definition specifies the name of a new function and the sequence of statements that execute when the function is called.

```
def print_message():
  print "CS101 is fantastic!"
  print "Programming is fun!"
```

You can call a function inside another function:

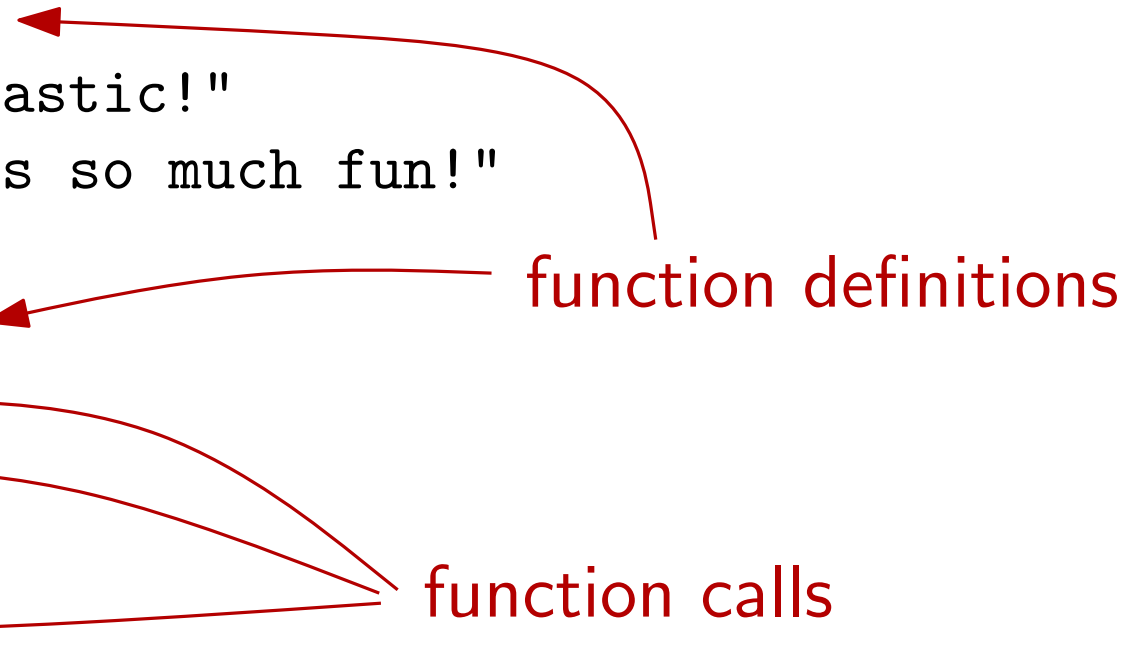
```
def repeat_message():
  print_message()
  print_message()
```

Flow of execution

```
def print_message():
  print "CS101 is fantastic!"
  print "Programming is so much fun!"

def repeat_message():
  print_message()
  print_message()

repeat_message()
```



Execution begins at the first statement. Statements are executed one-by-one, top to bottom.

Function definitions do not change the flow of execution—but only define a function.

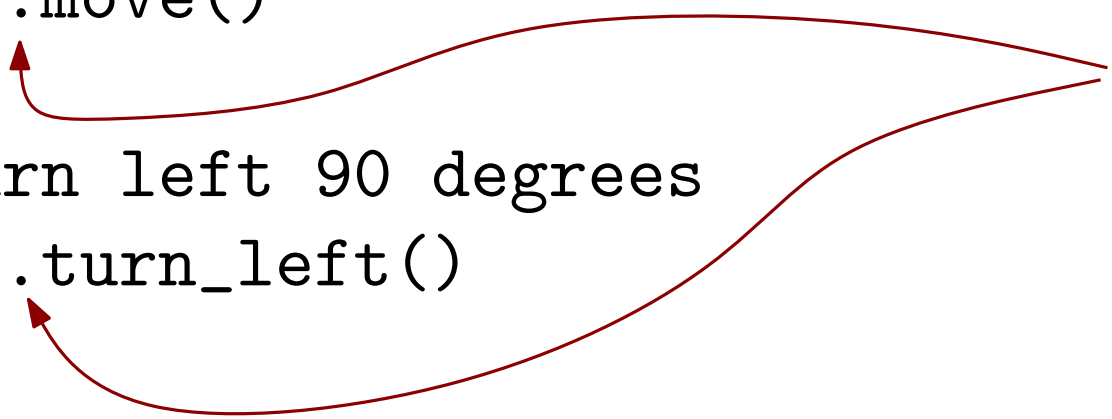
Function calls are like detours in the flow of execution.

Robots

```
# create a robot with one beeper
hubo = Robot(beepers = 1)

# move one step forward
hubo.move()

# turn left 90 degrees
hubo.turn_left()
```



dot notation

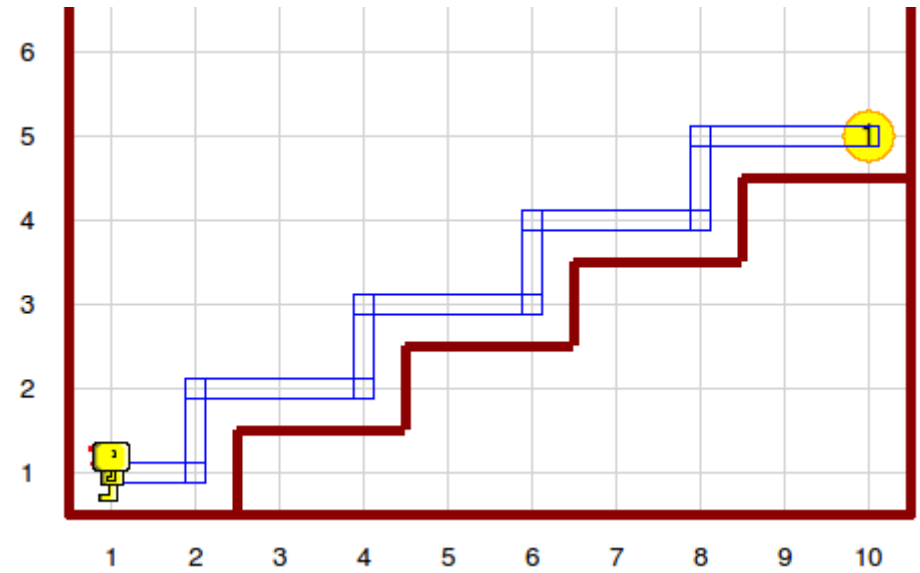
How can hubo turn right?

```
def turn_right():
  hubo.turn_left()
  hubo.turn_left()
  hubo.turn_left()
```

Define a function!

Newspaper delivery

Hubo should climb the stairs to the front door, drop a newspaper there, and return to his starting point.



Problem outline:

- Climb up four stairs
- Drop the newspaper
- Turn around
- Climb down four stairs

Python version:

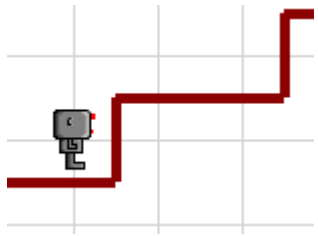
```
climb_up_four_stairs()
hubo.drop_beeper()
turn_around()
climb_down_four_stairs()
```

Implementing the steps

```
def turn_around():
  hubo.turn_left()
  hubo.turn_left()

def climb_up_four_stairs():
  climb_up_one_stair()
  climb_up_one_stair()
  climb_up_one_stair()
  climb_up_one_stair()

def climb_up_one_stair():
  hubo.turn_left()
  hubo.move()
  turn_right()
  hubo.move()
  hubo.move()
```

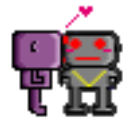


Top-down design

Start at the top of the problem, and make an outline of a solution.

For each step of this solution, either write code directly, or outline a solution for this step.

When all the partial problems have become so small that we can solve them directly, we are done and the program is finished.



Simple repetitions

To repeat the same instruction 4 times:

```
for i in range(4):
  print "CS101 is fantastic!"
```

for-loop

Don't forget the indentation!

What is the difference:

```
for i in range(4):
  print "CS101 is great!"
  print "I love programming!"
```

and

```
for i in range(4):
  print "CS101 is great!"
print "I love programming!"
```

Avoiding repetitions

```
def climb_up_four_stairs():
  climb_up_one_stair()
  climb_up_one_stair()
  climb_up_one_stair()
  climb_up_one_stair()
```

We should avoid writing the same code repeatedly. A `for`-loop allows us to write this more elegantly:

```
def climb_up_four_stairs():
  for i in range(4):
    climb_up_one_stair()
```