# Lecture 3: Decomposition & Refinement

Covered in this lecture:

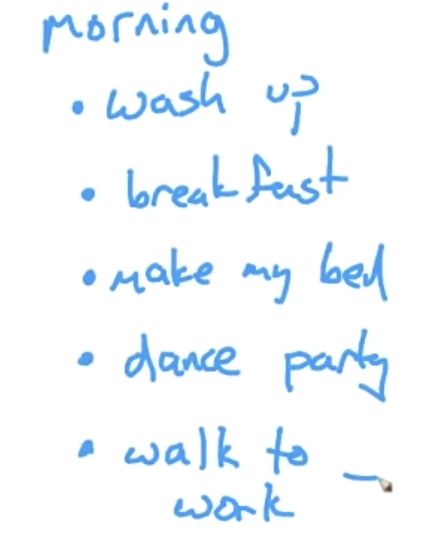
1. [Decomposition](#_gqitjkx9dhno)
2. [Mountain Example](#_auyv66sgdsbr)
3. [Roomba example](#_l2dkz8pyn2eq)
4. Wordsearch Example

**Lecture summary:** Students will learn about decomposition — breaking down large tasks into smaller, more manageable tasks. They will then apply decomposition to different Karel problems: Mountain, Rhoomba, and Wordsearch.

# Decomposition

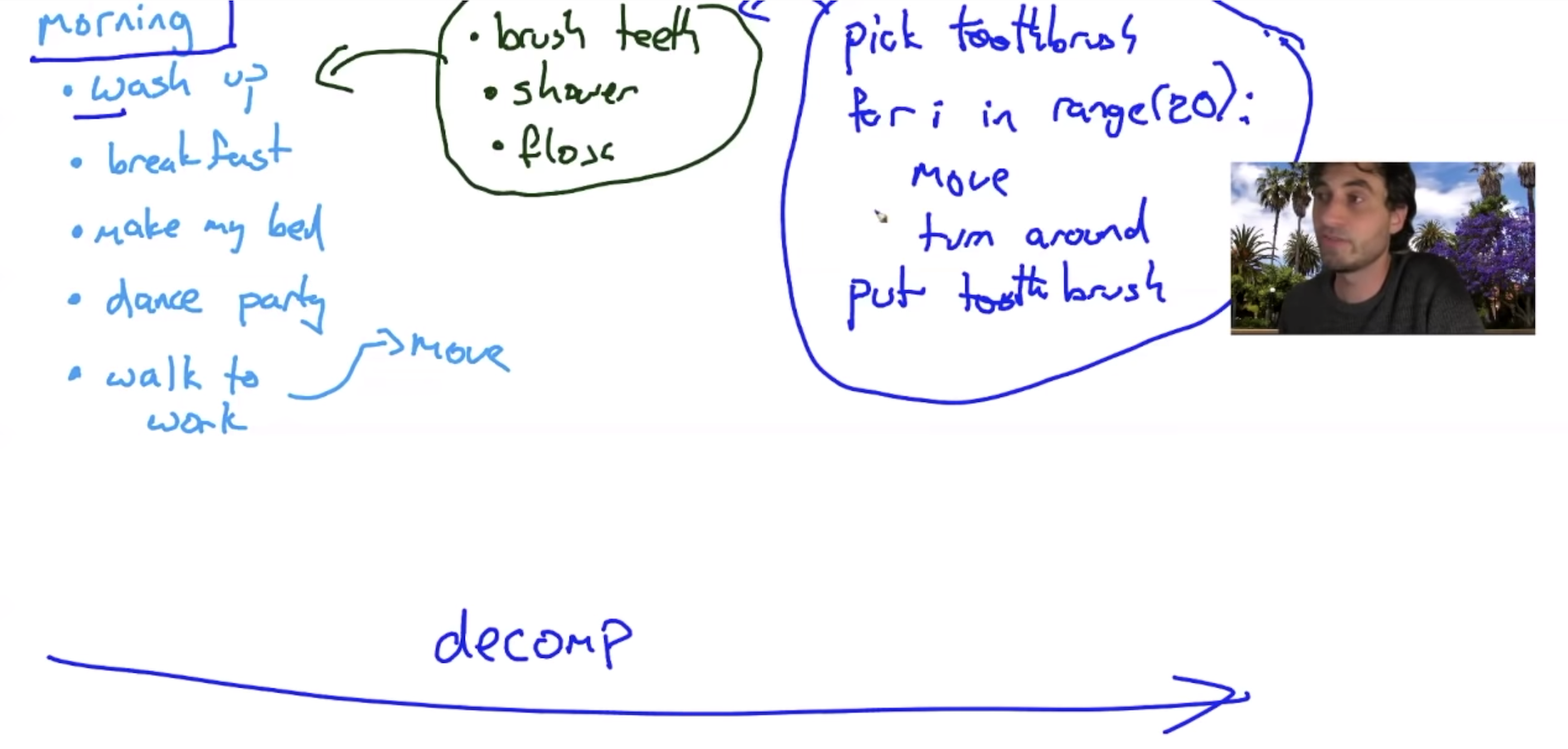
Decomposition means breaking down a large, complex task into smaller, manageable tasks. One example Chris illustrates this is describing his morning routine.

Chris’s morning routine:



*(alt: Chris’s morning routine as a list of tasks: wash up, breakfast, make my bed, dance party, walk to work)*

Note how when Chris describes his morning routine, he’s not concerned about the exact number of steps he needs to walk from his bed to his bathroom, or what tasks he needs to do to wash up. This list is just a high-level overview of what he needs to do — breaking down his morning routine into manageable tasks.



*(Alt: Adding on to the list of Chris’s tasks for his morning routine, we can focus in on the tasks for “wash up.” Chris lists that to wash up, he must: brush teeth, shower, floss. He then writes what the code might look like for brushing his teeth:*

pick toothbrush

for i in range(20):

move()

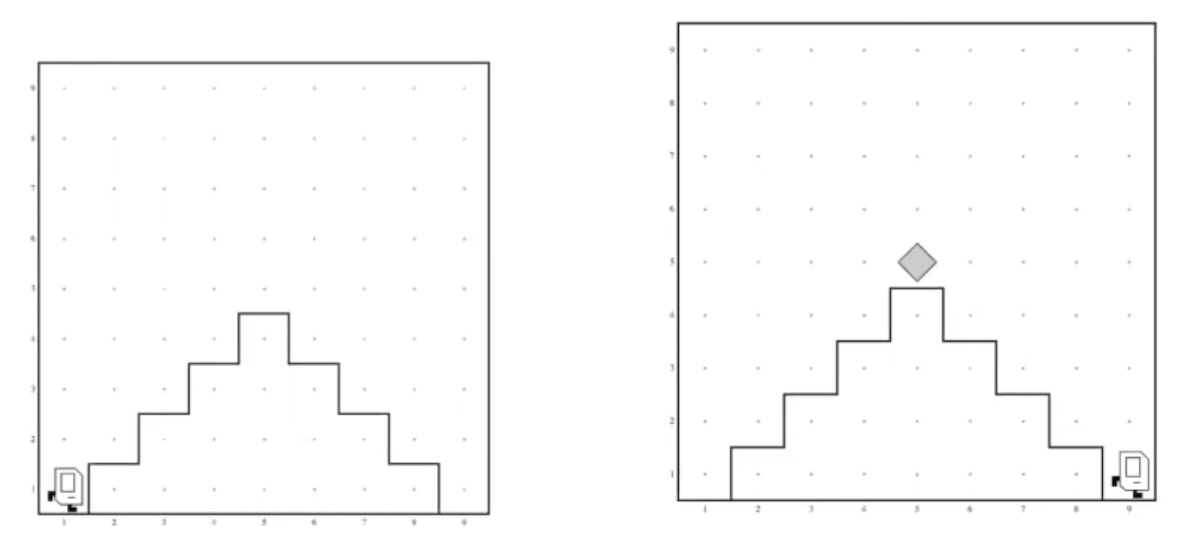
turn\_around()

put toothbrush*)*

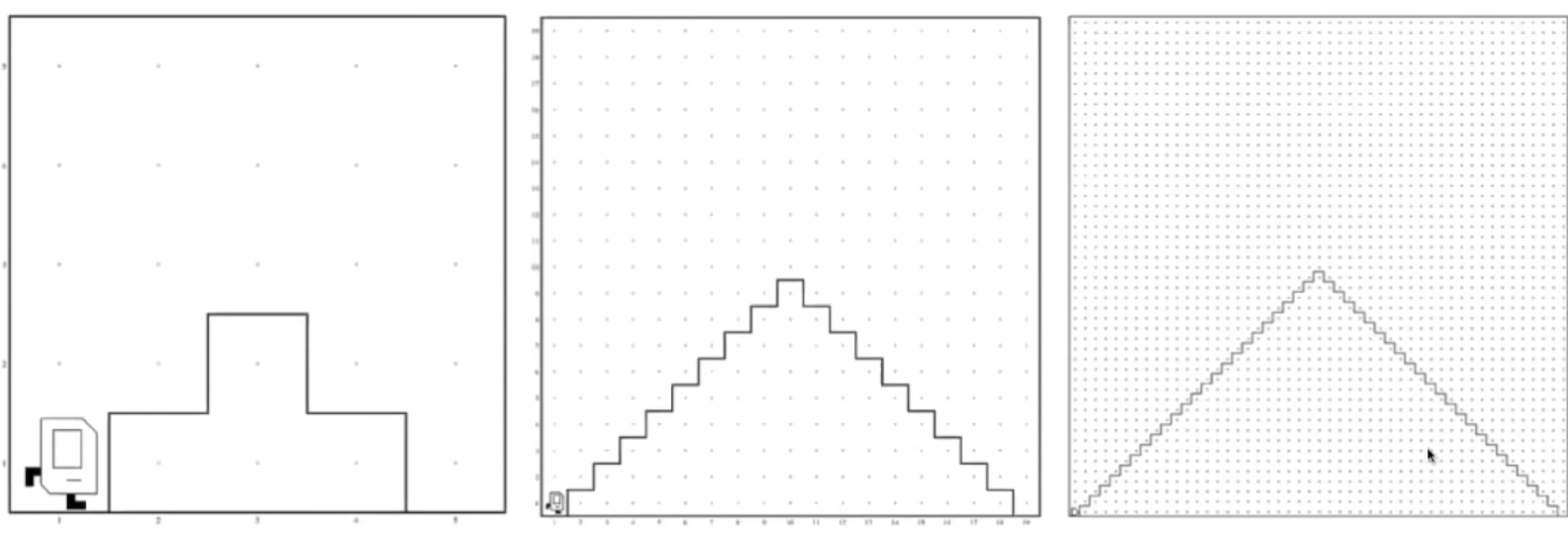
If we want to go into more detail, we can look at what tasks make up washing up: brush teeth, shower, floss. We can go into even more detail and think about what the code for brushing Chris’s teeth looks like. The takeaway is that before thinking about code, take a step back to look at the bigger picture: what are the tasks that need to be accomplished, and how are the tasks related to each other?

# Mountain Example

In this program, we want Karel to climb a mountain from the left side, plant a beeper at the top of the mountain, and walk down the mountain on the right side



The mountains can be of any size:



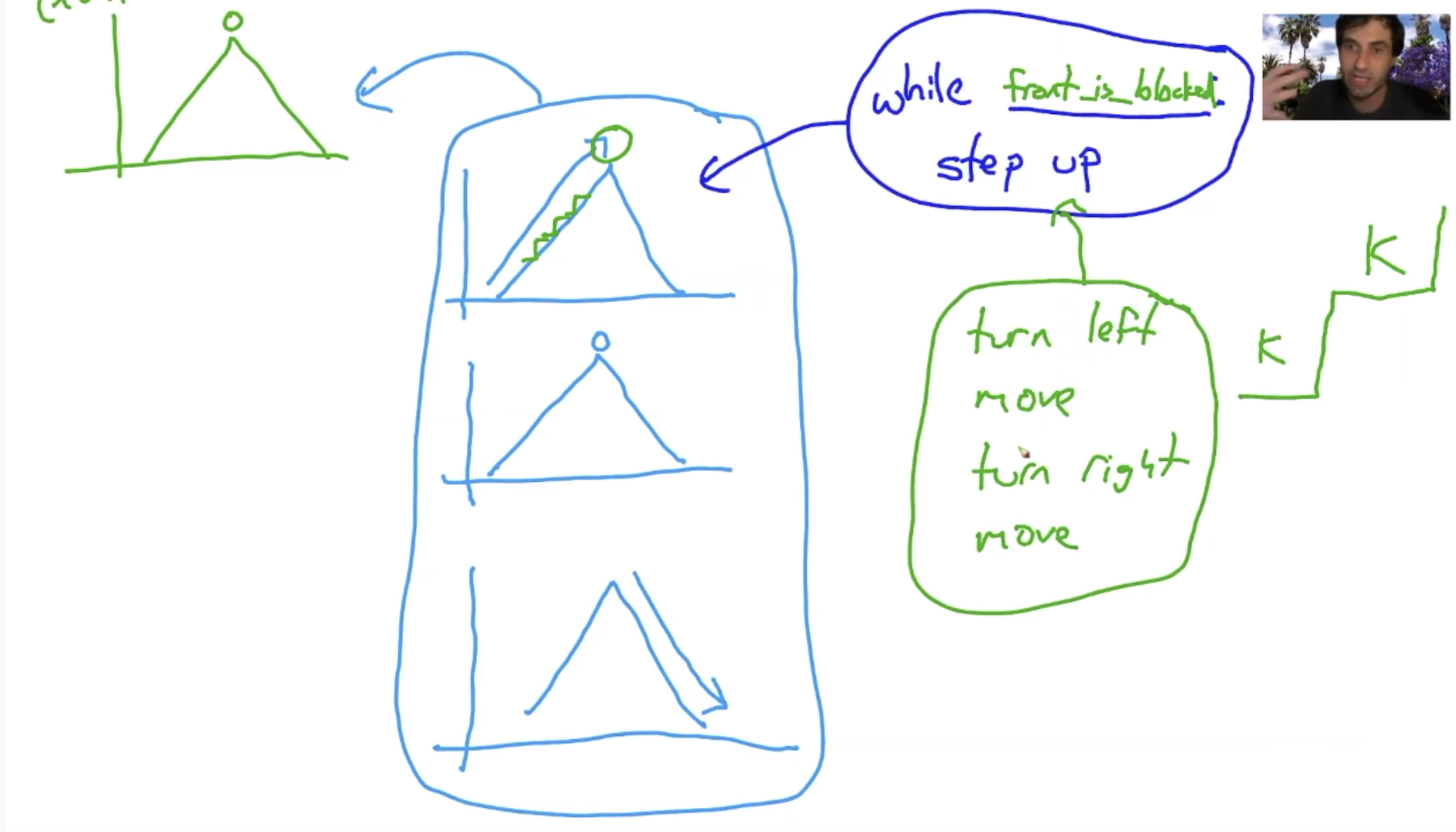
**How can we break this problem down into smaller parts?**

We can look at this problem as three parts:

|  |  |
| --- | --- |
| 1 | Ascend the mountain |
| 2 | Plant beeper at the top of the mountain |
| 3 | Descend the mountain |

Now that we’ve split the program into 3 parts, we can delve into what writing one part of the program would look like.

Here’s how we would do part 1: ascend the mountain:



**Now, let’s take a look at how we would write our code:**

def main()

ascend\_mountain()

put\_beeper()

descend\_mountain()

#pre: karel is facing a mountain

#post: karel is on top of said mountain!!

def ascend\_mountain():

while front\_is\_blocked():

step\_up()

#pre: karel is facing a step of the mountain

#post: karel is one step up also facing the mountain

def step\_up():

turn\_left()

move()

turn\_right()

move()

def turn\_right()

for i in range(3):

turn\_left()

**Next, we will need to write our code for** descend\_mountain():

def descend\_mountain():

while front\_is\_clear():

step\_down()

def step\_down():

move()

turn\_right()

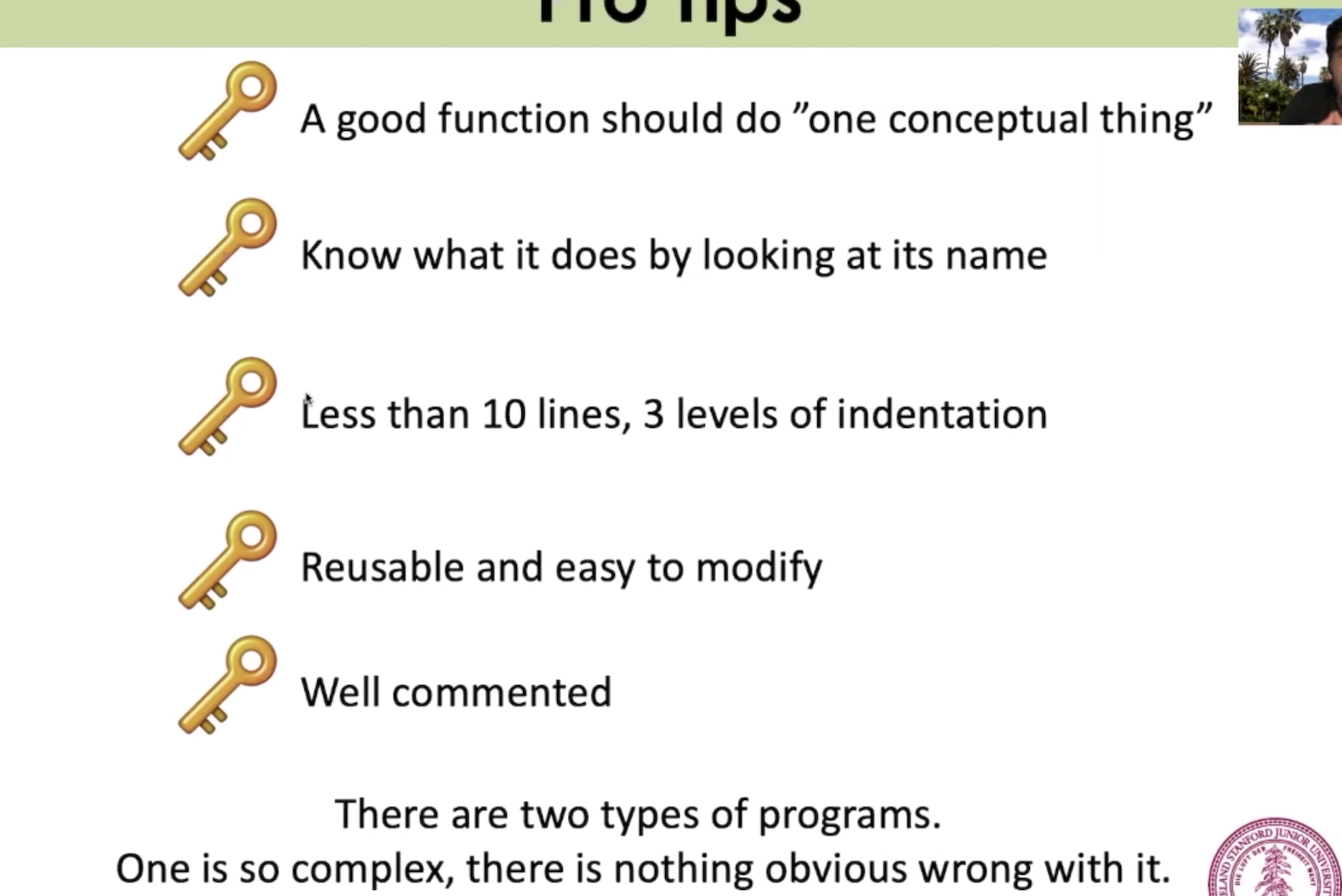
move()

turn\_left()

The following two programs do the same thing: have Karel climb a mountain. Notice that while the code on the left is shorter, it’s hard to understand what Karel is doing. With the code on the right, we can better understand the tasks Karel is completing.

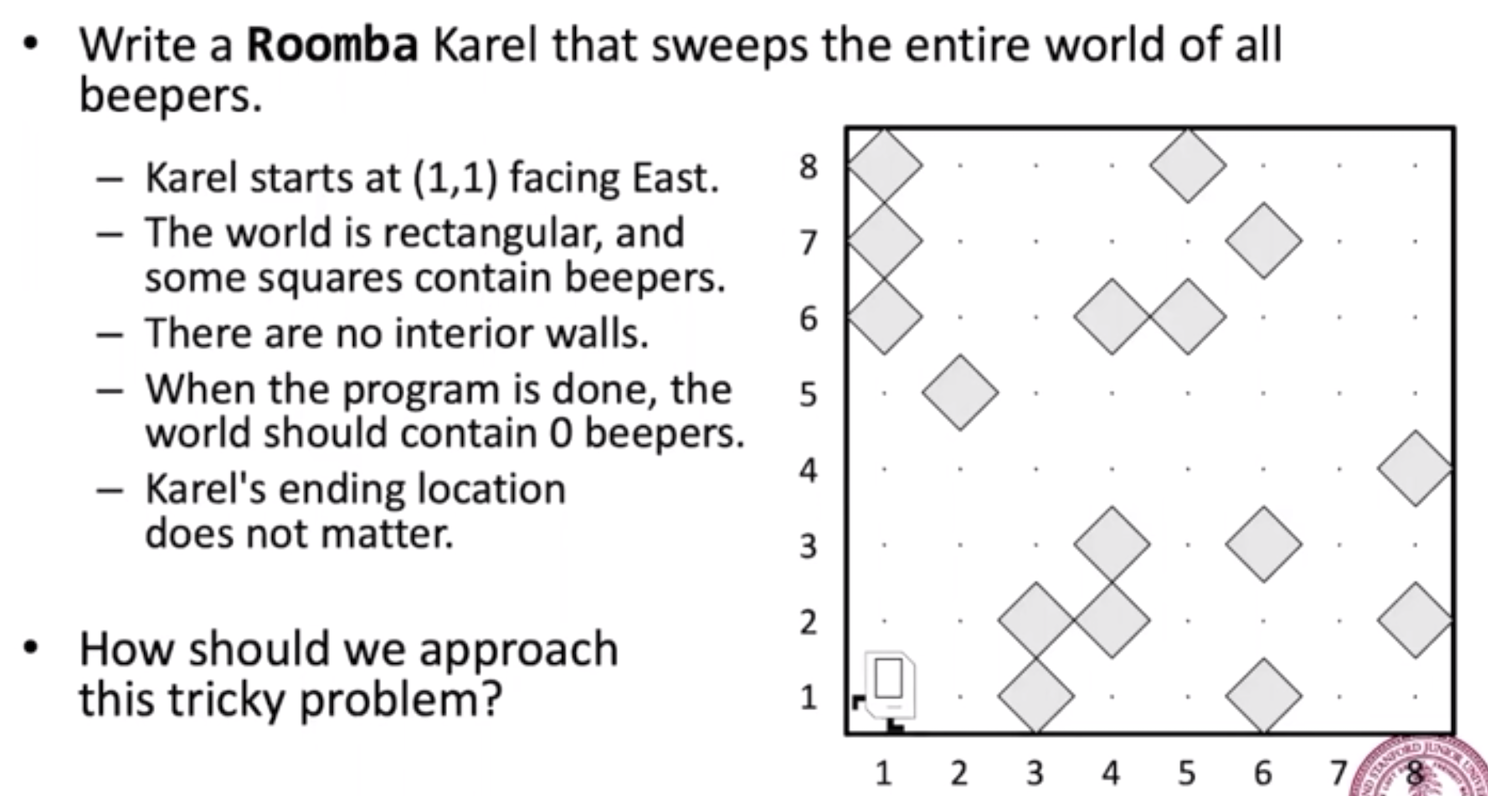
|  |  |
| --- | --- |
| def main():  while front\_is\_blocked():  turn\_left()  move()  turn\_left()  turn\_left()  turn\_left()  move()  put\_beeper()  while front\_is\_clear():  move()  turn\_left()  turn\_left()  turn\_left()  move()  turn\_left() | def main():  ascend\_mountain()  put\_beeper()  descend\_mountain()  #pre: karel is facing a mountain  #post: karel is on top of said mountain!!  def ascend\_mountain():  while front\_is\_blocked():  step\_up()  def descend\_mountain():  while front\_is\_clear():  step\_down()  def step\_down():  move()  turn\_right()  move()  turn\_left()  #pre: karel is facing a step of the mountain  #post: karel is one step up also facing the mountain  def step\_up():  turn\_left()  move()  turn\_right()  move()  def turn\_right()  for i in range(3):  turn\_left() |

Pro-tips on what makes a good function:



# Roomba Karel

We want to write a program where Karel will clean up all the beepers in its world:



In this program, Karel will need to touch every spot in the world. There are a few ways Karel can do this:

|  |  |
| --- | --- |
| Algorithm 1: Spiral | Algorithm 2: Zig Zag |
| Algorithm 3: | Algorithm 4: Comb |

Most students will initially pick Algorithm 2 because it minimizes the number of steps Karel will have to take. Chris tells students that 99% of the time, it is better to have a program that works correctly than a program that uses the fewest number of steps.

In both Algorithm 2 and Algorithm 4, Karel is repeating the same process of moving across a row. There is one powerful property about the way Algorithm 4: Comb works — Karel always has the same pre and post conditions. With consistent pre and post conditions, it’s easier for us to write helper functions to clear each row.

Here’s the code for Roomba Karel:

|  |
| --- |
| def main():  while left\_is\_clear():  #Pre: Karel is facing East, start of row  clear\_row()  next\_row()  #Post: Karel is facing East, start of next row  clear\_row()  #pre: karel is at the start of a messy row  #post: karel is at the end of a clean row  def clear\_row():  while front\_is\_clear():  safe\_pick()  move()  safe\_pick()  #pick up, but only if there is a beeper  def safe\_pick():  if beepeers\_present():  pick\_beeper()  def next\_row():  turn\_around()  move\_to\_wall()  turn\_right()  move()  turn\_right()  def move\_to\_wall():  while front\_is\_clear():  move() |

# Lecture 4: Intro to Python

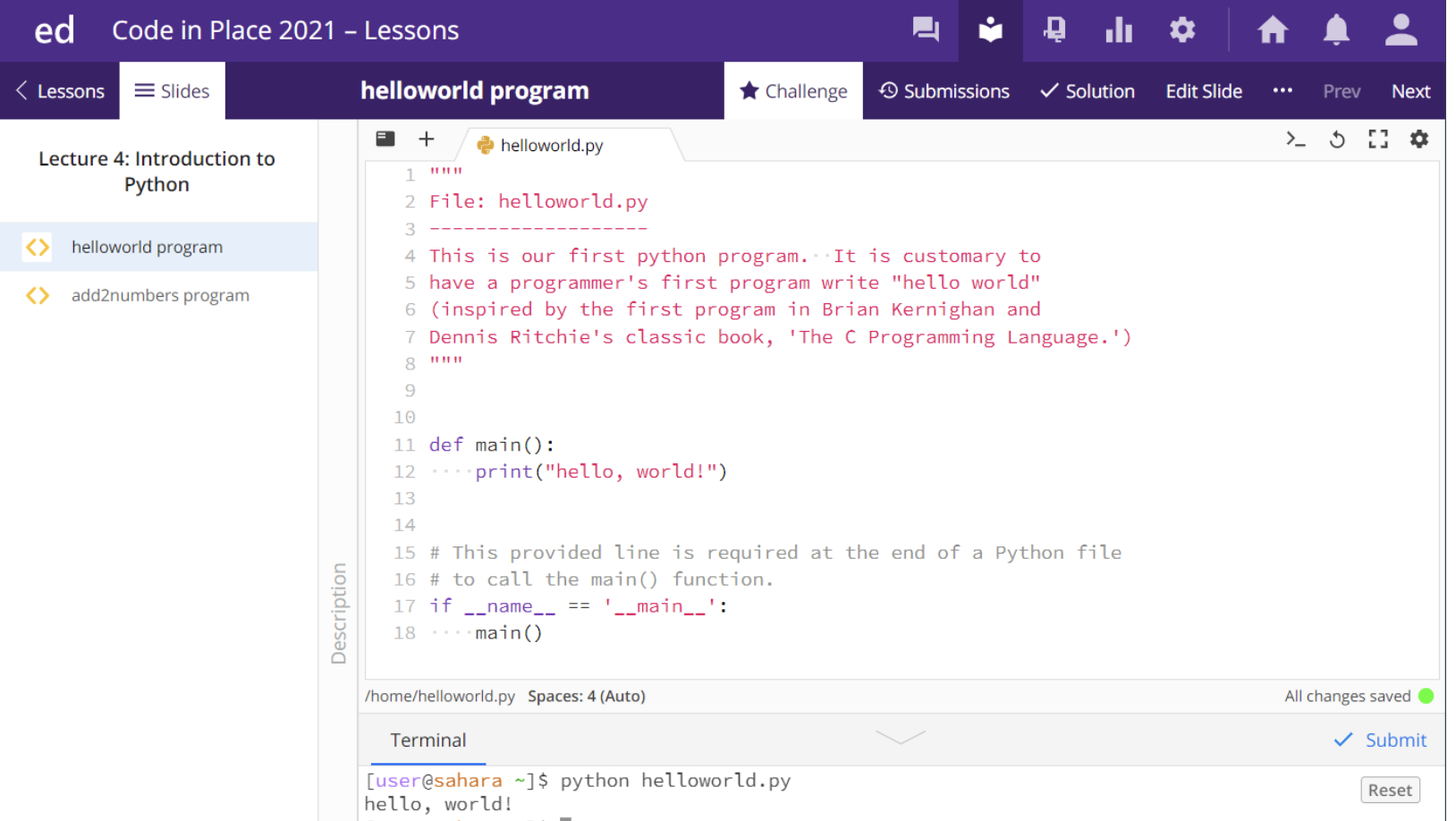
(Bye, bye Karel!)

Covered in this Lecture:

1. HelloWorld
2. [Add2Num: Understanding variables](#_a73bo865qvft)

**Lecture summary:** Students have graduated from Karel and will being working with real-world Python. This lecture covers print statements, getting input, and variables.

# HelloWorld



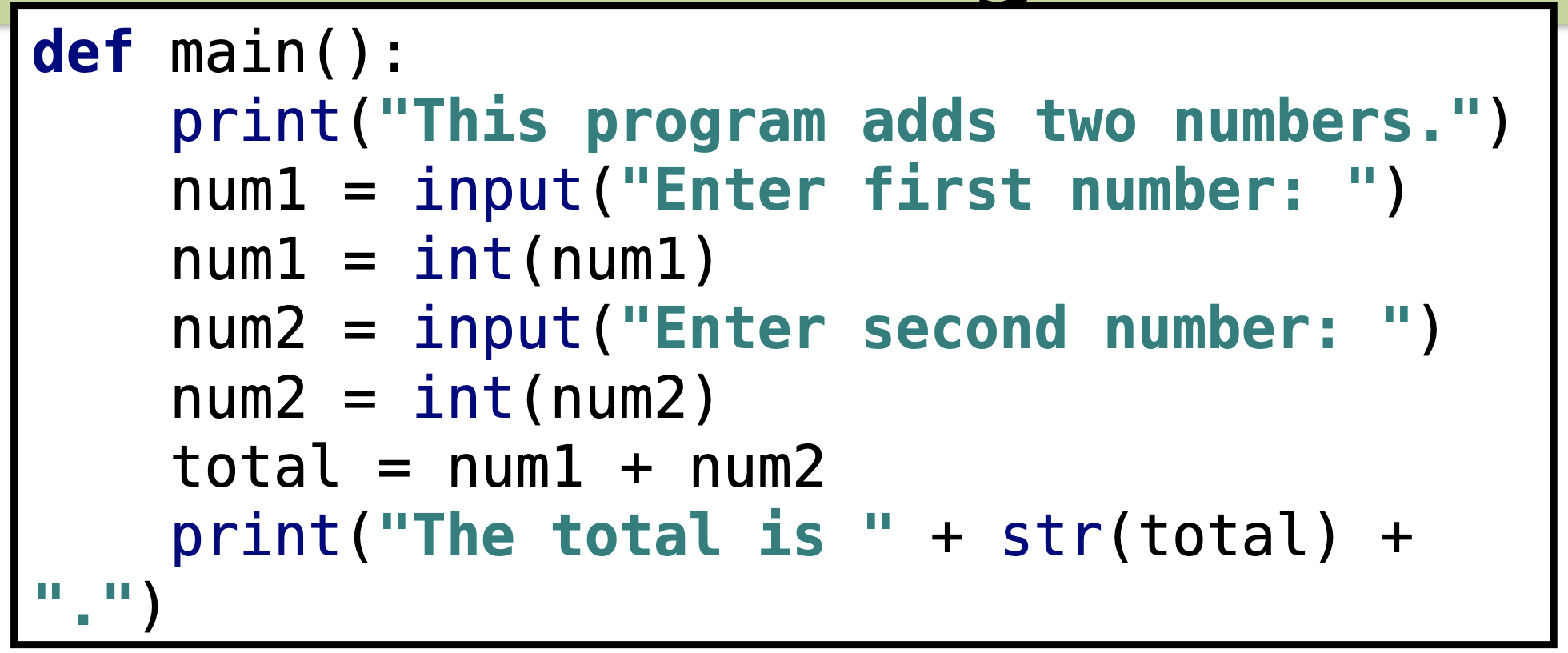
Students get a chance to write their first python program: Hello world!

To run a Python program in Ed, click on the terminal and type: python helloworld.py

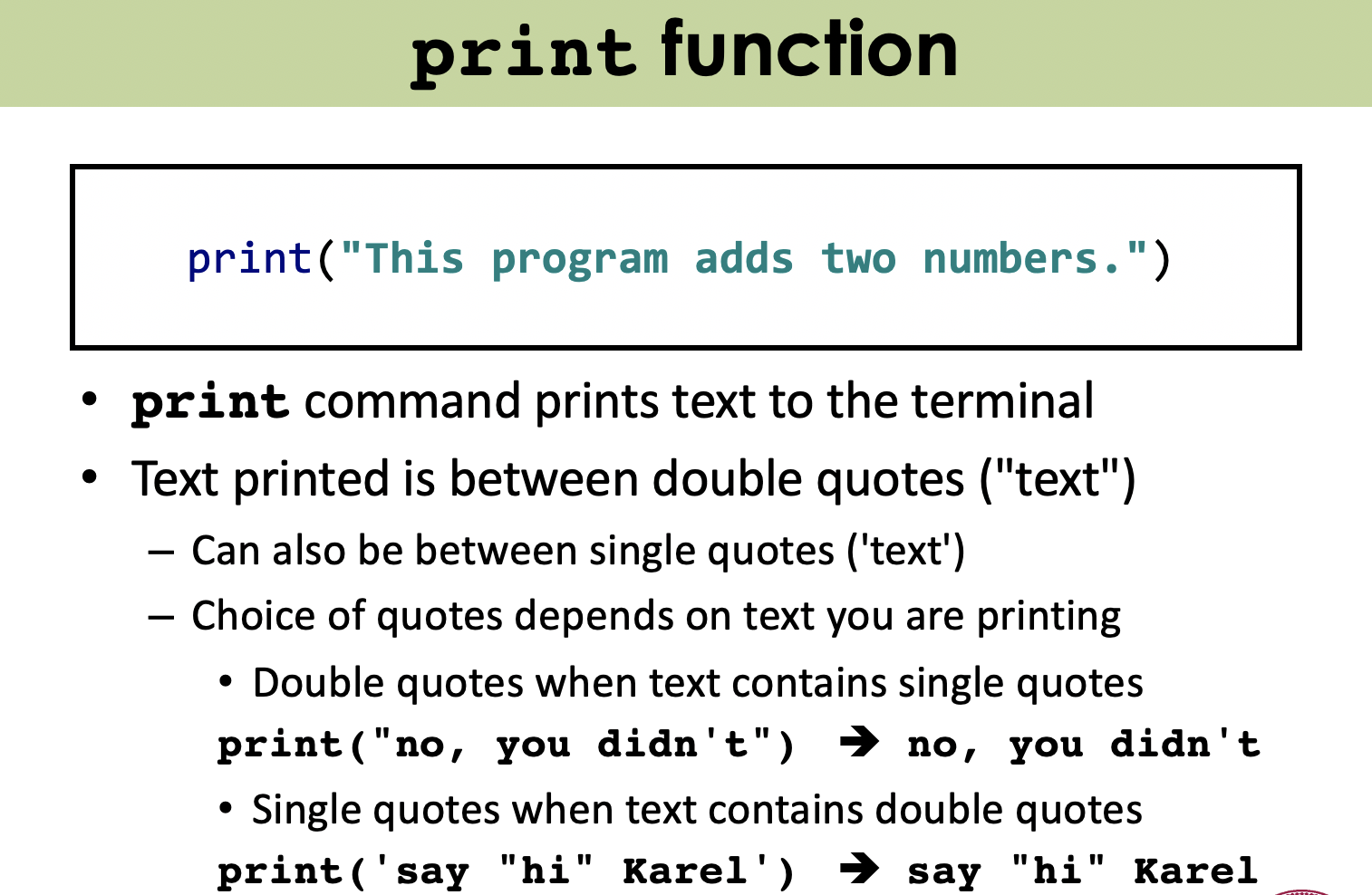
(python filename.py)

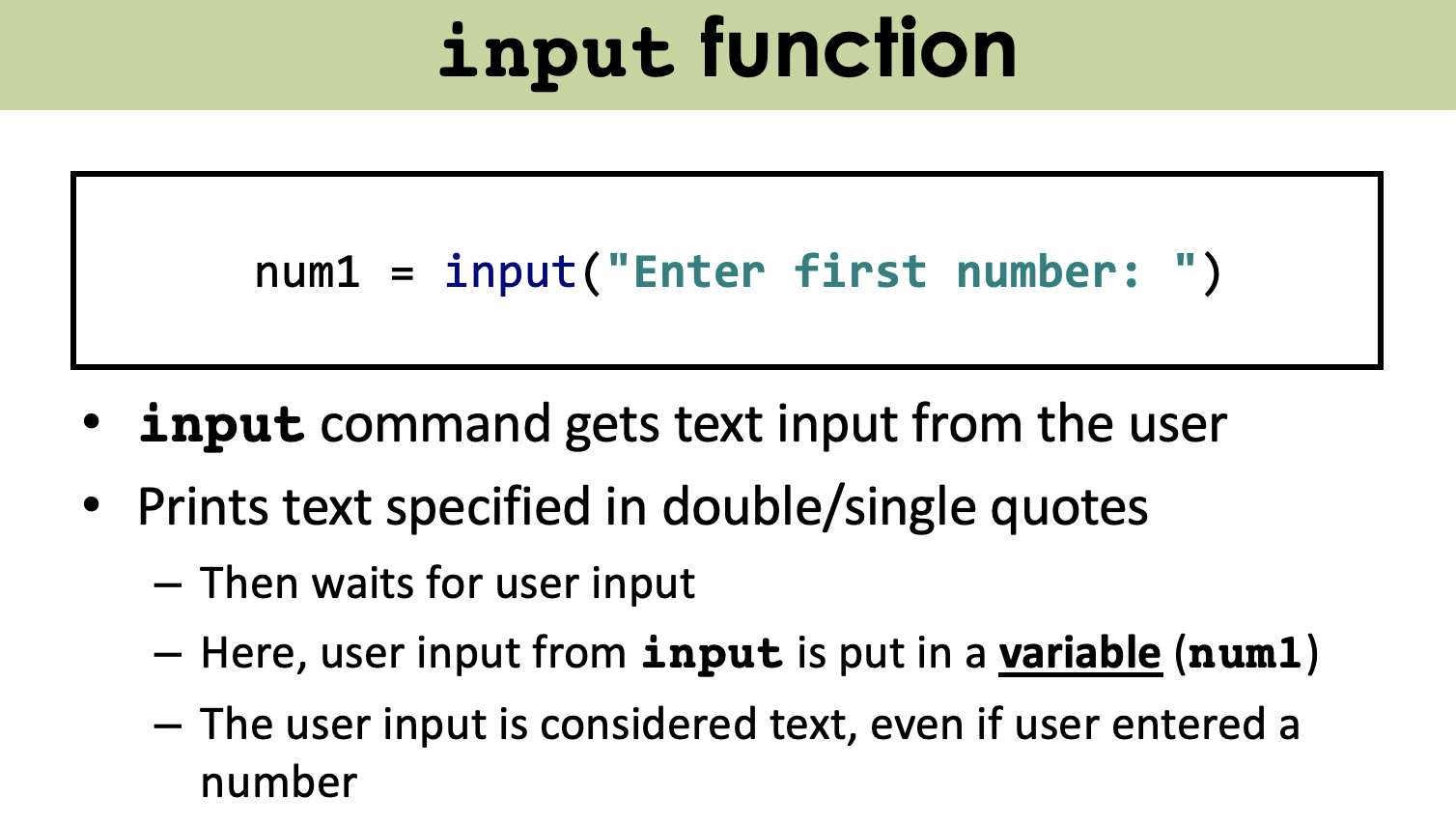
# Add2Num

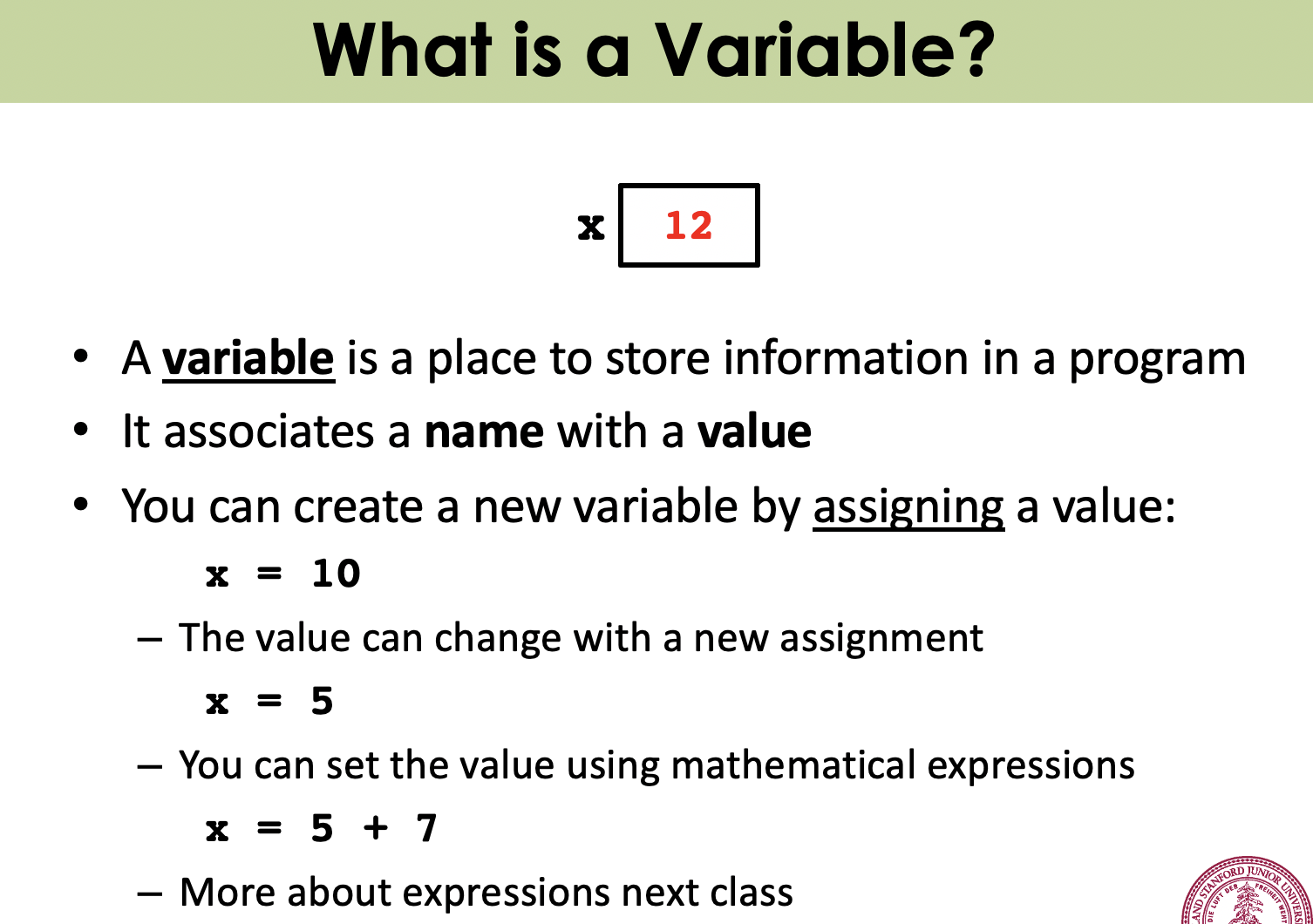
Here is the Add2Num Program, which takes 2 numbers as input from a user and prints out the sum of those 2 numbers:

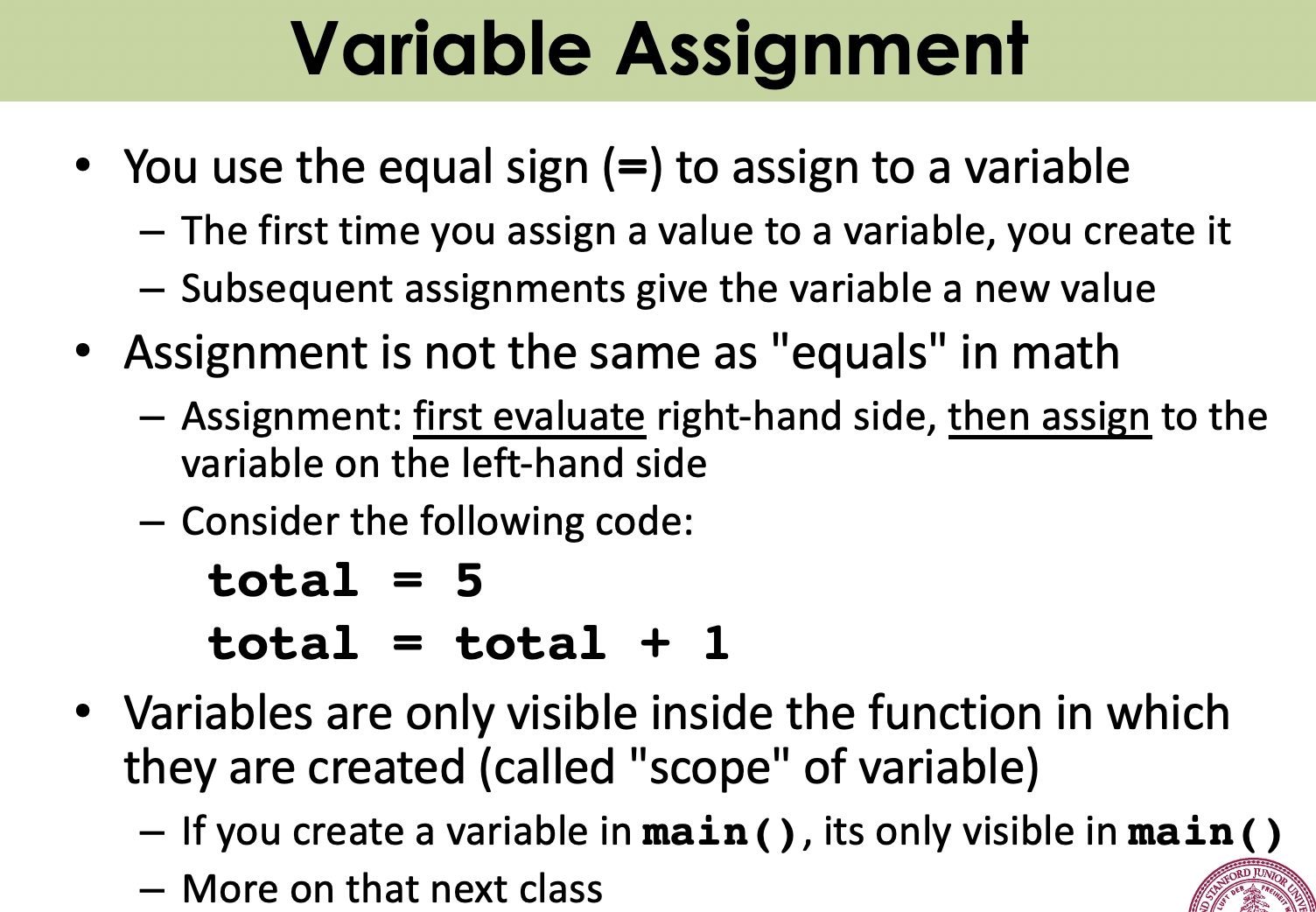


What does each line mean?

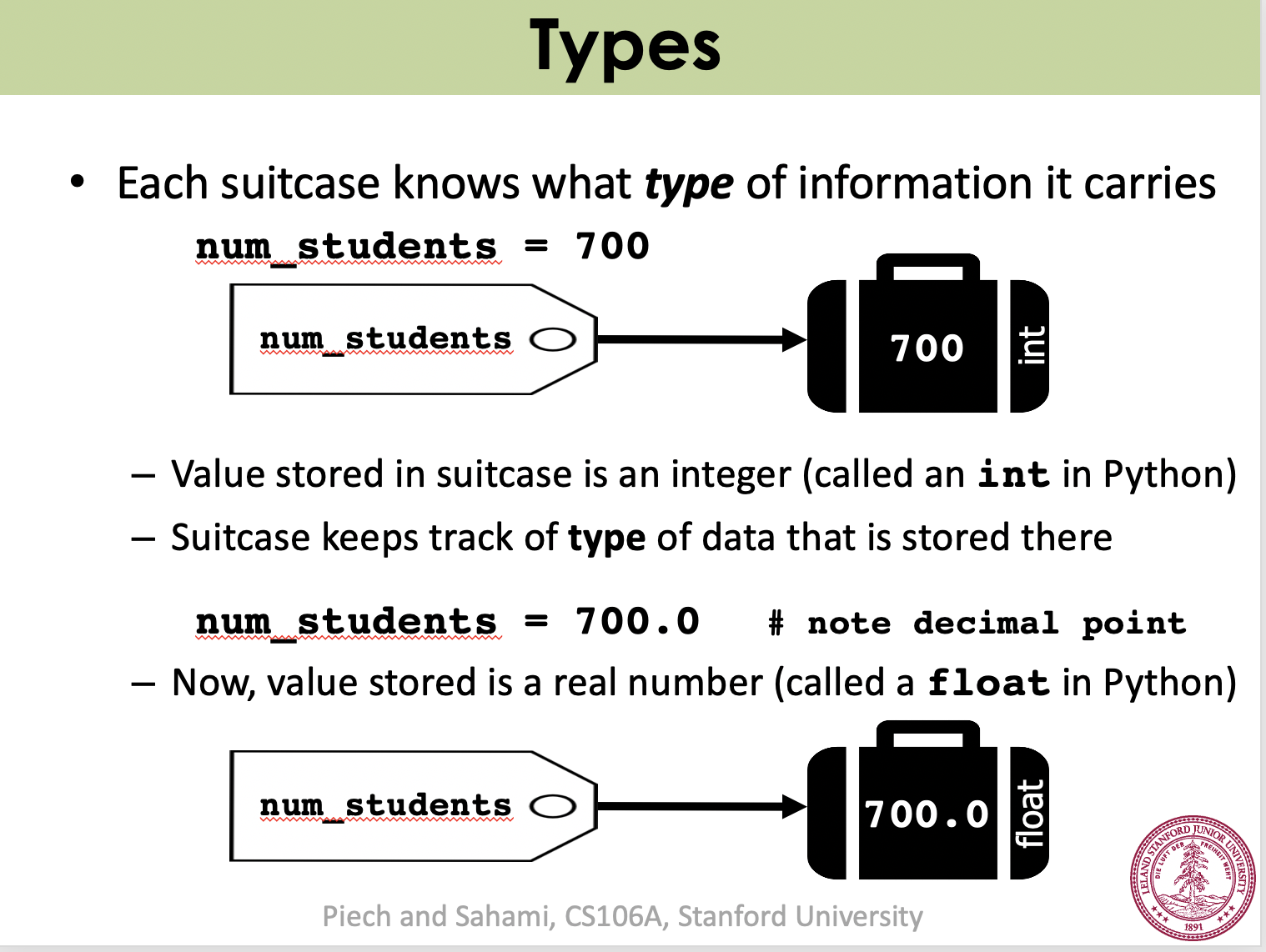










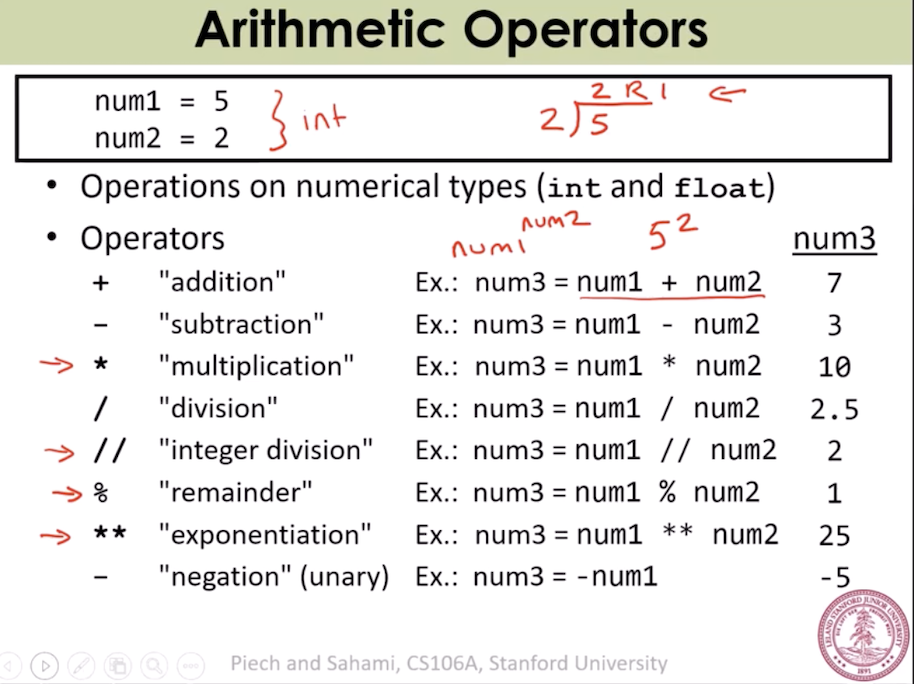


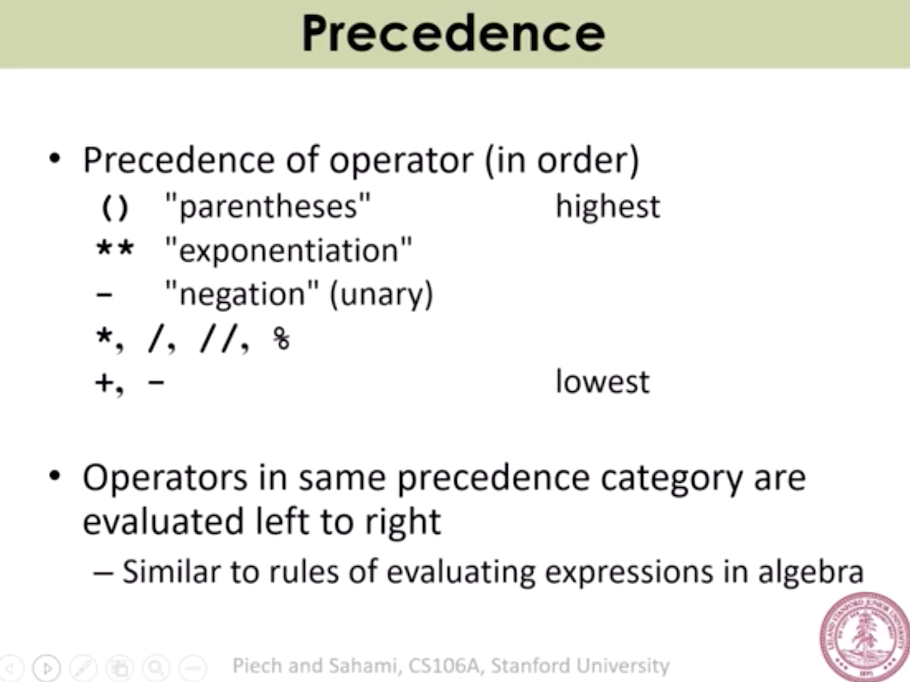
# Lecture 5: Expressions

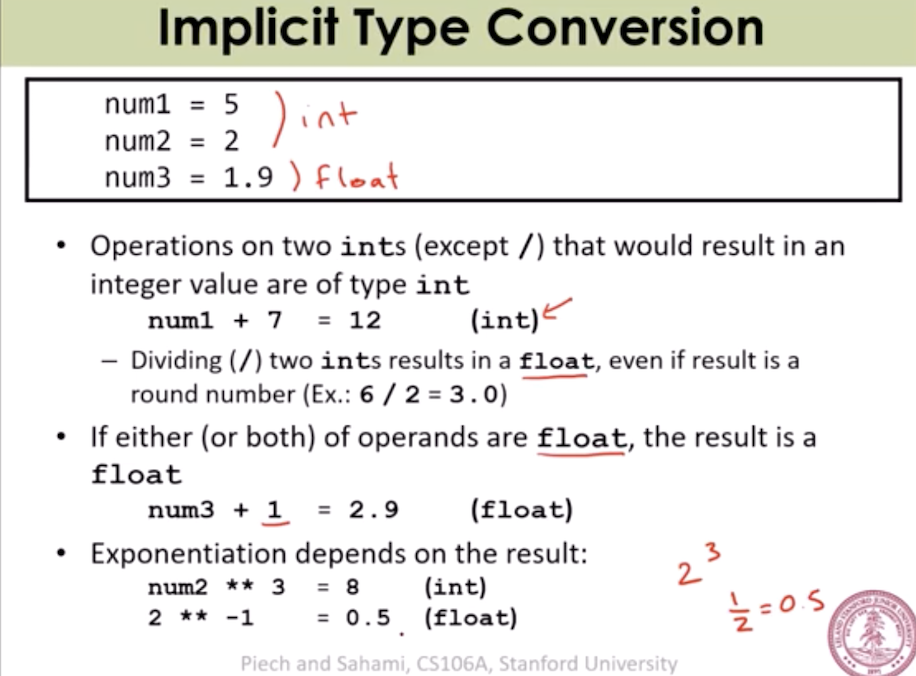
Covered in this lecture:

1. [Expressions](#_lvgz0nqehnu3)
2. [Constants](#_we0fgackccjm)
3. [Math library](#_yf0tqjwob25y)
4. [Random numbers](#_mveas4sp975w)

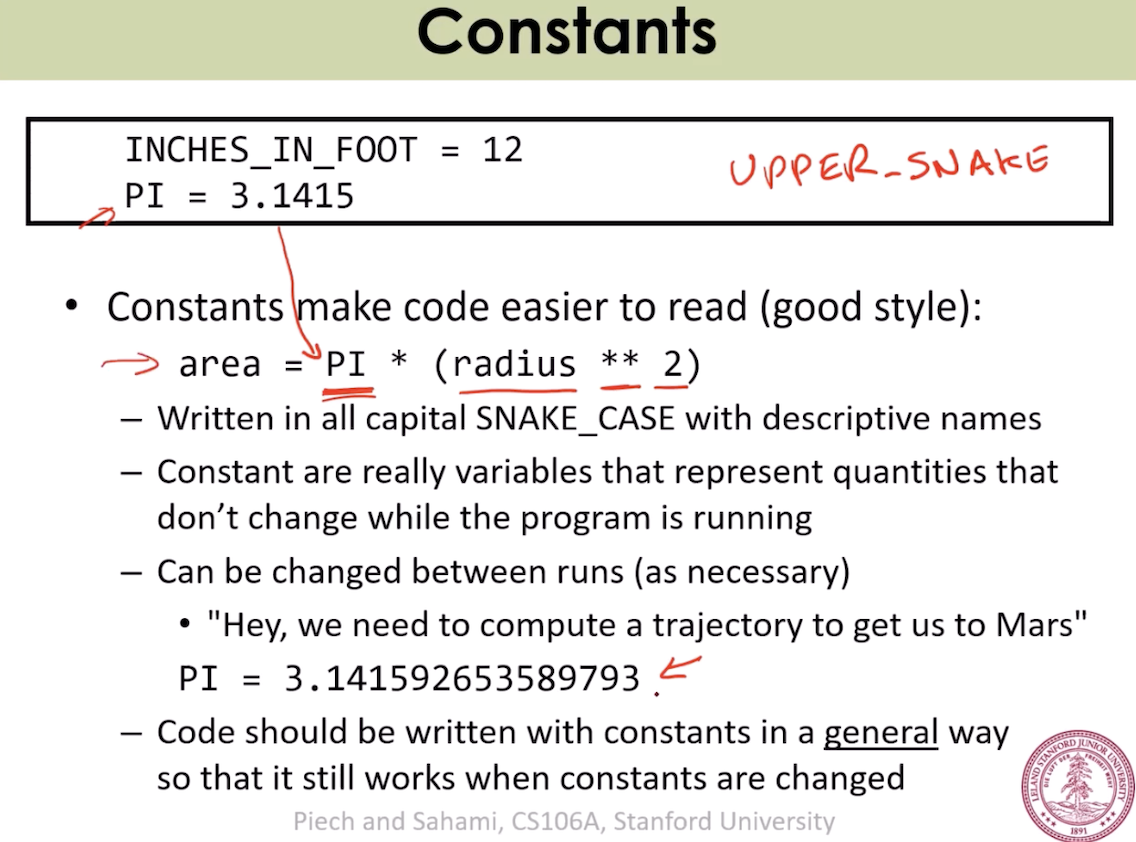
# Expressions



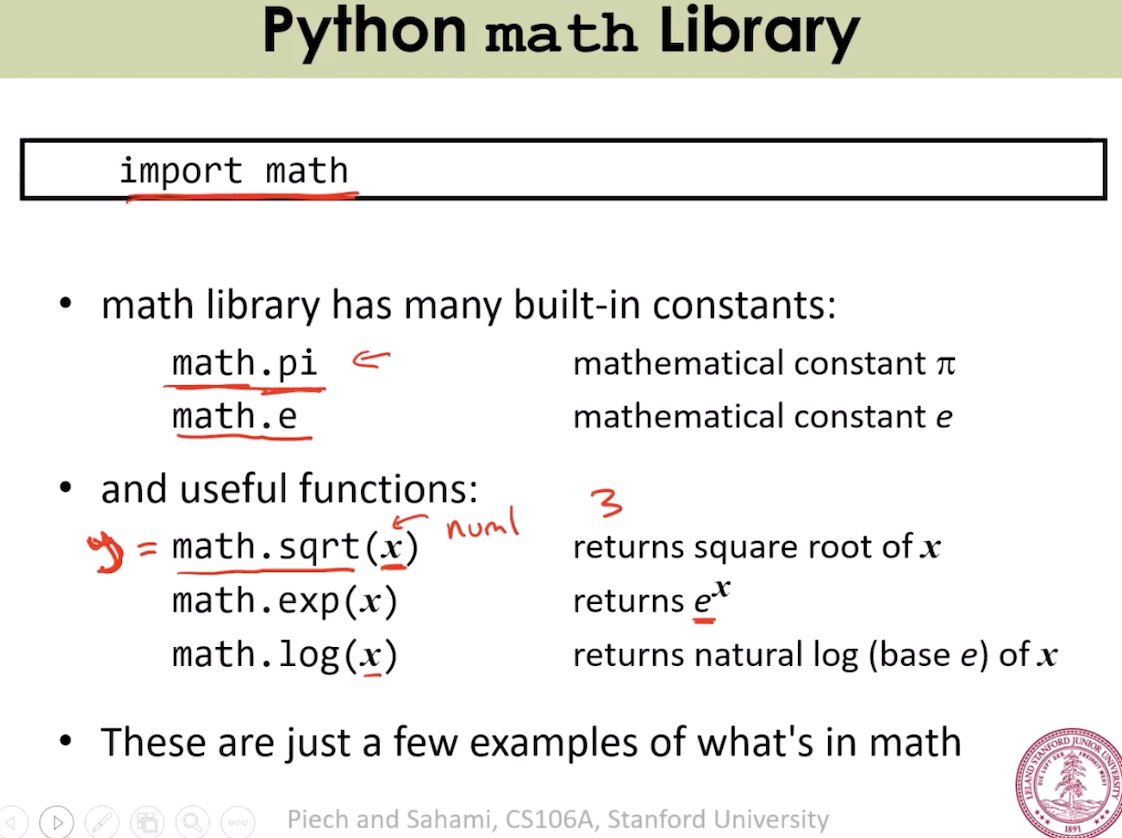




# Constants



# Math library

’

# Random numbers

