

Coding For Creative Robotics

Week 3: Computational Thinking, Conditionals &
Debugging


Agenda

Lecture (90'):


- Last week recap
- Computational Thinking?
- Control Flow
- Conditionals
- Debugging

Tutorial (90'):


- Homework questions
- Conditionals exercises




DMs



Activity




Later



More

Technicians

 **staff-technicians**

▼ Channels

bsc-cr-one

bsc-cr-two


marketplace


msc-cc-23


msc-cr

msc-cr-23

research-ethics


 staff-robotics

 staff-technical-pcomp-kits



 staff-technicians-checkout


technical-skills

thelink




Congratulations [@Haofei](#) and [@Ksenia](#) 🎉🎊

 3 



Rohit 10:21

For those who are looking for this weeks' coding homework, please find it below



Rohit

Hi All,

Sonia has prepared some homework for you all to try.

Please download the attached ipynb file and follow the instructions to complete the homework.



Binary ▼

01

Creative_Robotics_W2_Tutorial_Homework.ipynb

Binary

Posted in # msc-cr | 14 Oct | [View message](#)


 1 

Week3 Lecture Slides

Tomorrow's lecture slides have been uploaded on Moodle.

Share an item?



 is **requesting access** to the following item:

 Creative_Robotics_W2_Tutorial_Homework_solution.ipynb

Manage sharing

Share an item?



 is **requesting access** to the following item:

 Creative_Robotics_W2_Tutorial_Homework_solution.ipynb

Manage sharing

Access denied!

Last week recap

A computer is like a violin. You can imagine a novice trying first a phonograph and then a violin. The latter, he says, sounds terrible. That is the argument we have heard from our humanists and most of our computer scientists.

Computer programs are good, they say, for particular purposes, but they aren't flexible. Neither is a violin, or a typewriter, until you learn how to use it.

—Marvin Minsky, “Why Programming Is a Good Medium for Expressing Poorly-Understood and Sloppily-Formulated Ideas”

Computational Thinking

"algorithmic thinking"

"think computationally"

"think like a computer scientist"

What does it even mean?

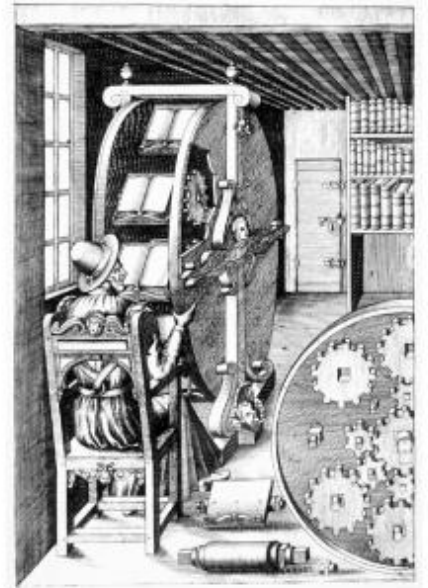


Computational Thinking

“Every computer program is a model, hatched in the mind, of a real or mental process.”

Free online: <https://web.mit.edu/6.001/6.037/sicp.pdf>

Structure and Interpretation of Computer Programs



SECOND EDITION

Unofficial Texinfo Format 2.andresraba5.6

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman
foreword by Alan J. Perlis

Algorithm or a Program ?

Algorithm or a Program

- **Algorithm:** A step-by-step set of operations to solve a specific problem. Algorithms focus on the **process**, whereas a **program** is the **implementation**.

- **Seminal Work on Algorithms:**

-

We look to **MIT's pioneering work in the 1960s and 1970s** on algorithm theory. The seminal text, *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein, and *Structure and Interpretation of Computer Programs* (SICP) by Abelson and Sussman, revolutionized how we think about breaking down problems and creating formal methods to solve them.

Computational Thinking

Algorithms are the ideas behind computer programs.

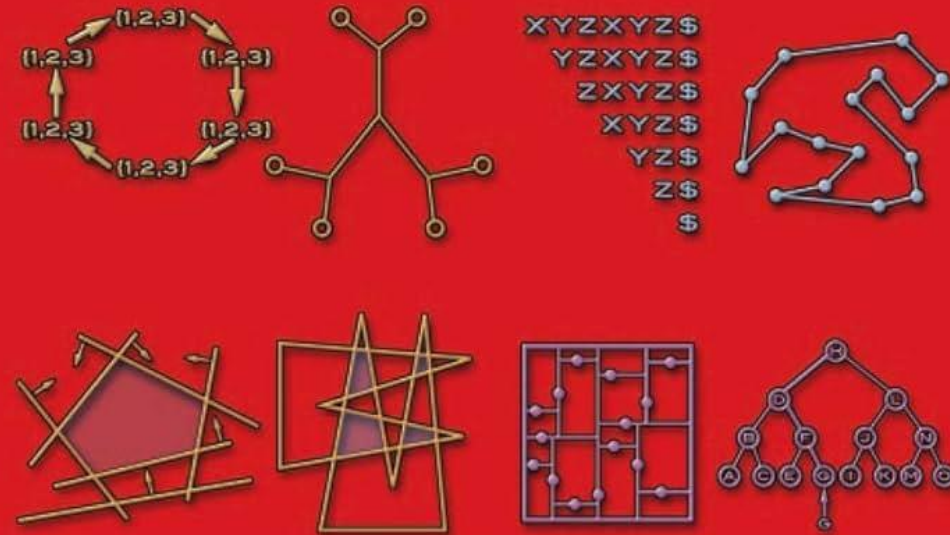
To be interesting, an algorithm has to solve a general, specified problem.

An algorithmic problem is specified by describing the set of instances it must work on, and what desired properties the output must have.

<https://www.algorist.com/>

Second Edition

THE Algorithm Design MANUAL



Steven S. Skiena

 Springer

Computational Thinking

The single most important skill for a computer scientist is problem solving.

Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills.

<https://openbookproject.net/thinkcs/python/english3e/#>



Learning with Python 3 (RLE)

Version date: October 2012

by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers

Control Flow #1

How we control decision making?

"Control flow is how we direct the execution of a program. As discussed in SICP, controlling the flow of a program effectively allows us to create complex, intelligent behaviour out of simple, deterministic steps.

Control flow can take many forms, including **sequential**,
decision-based,
and **repetition-based**.

Control Flow #2

Sequential Execution

The simplest form of control flow is sequential execution. This means that each line of code runs one after the other, from top to bottom, without any branching or looping.

```
print("Step 1: Start program")  
print("Step 2: Execute main function")  
print("Step 3: End program")
```

Control flow can take many forms, including **sequential**, **decision-based**, and **repetition-based**.

Control Flow #2

Conditional Statements

In Python, decision-based control flow is implemented through conditional statements like **if**, **elif**, and **else**. These structures let us branch the flow of our program and are fundamental to making our code more intelligent and responsive

Details in a minute!

Control flow can take many forms, including **sequential**, **decision-based**, and **repetition-based**.

Control Flow #3

Loops

Repetition-based control flow, also known as looping, allows us to repeat a block of code multiple times.

Python provides two primary types of loops: **for** loops and **while** loops. Loops help us simplify our code and reduce redundancy, making our programs more efficient and easier to manage.

More on this one next week!

Control flow can take many forms, including **sequential**,
decision-based,
and **repetition-based**.

Conditional Statements

if, elif (else if), and else

decision-based control flow

if Statements

The if statement is the simplest and most powerful tool we have for decision-making in code.

According to SICP, branching is a fundamental way that programs respond to changes in their environment or internal state.

With if statements, we execute a code block only if a specific condition is true. This allows our programs to make decisions dynamically.

```
temperature = 28  
if temperature > 25:  
    print("It's warm today.")
```

Pay attention to the tab here!

if Statements

```
if x % 2 == 0:    The Boolean expression after the if statement is called the condition.
    print(x, " is even.")
    print("Did you know that 2 is the only even number that is prime?")
else:
    print(x, " is odd.")
    print("Did you know that multiplying two odd numbers " +
          "always gives an odd result?")
```

if Statements

```
if x < 0:
```

```
    print("The negative number ", x, " is not valid here.")
```

```
    x = 42
```

```
    print("I've decided to use the number 42 instead.")
```

```
print("The square root of ", x, "is", math.sqrt(x))
```

else and elif

Chained conditionals

else and elif provide additional control over program execution.

elif allows for multiple conditions, while else runs code when no previous conditions are met.

According to Skiena, building logical decision trees with conditionals is key to defining program behaviour efficiently, without redundancy.

elif: Checks additional conditions if previous ones were not true.

else: Provides a fallback action, ensuring the program has a deterministic outcome.

```
score = 85
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
else:
    print("Grade: C")
```

Nested Conditionals

Nested conditionals allow us to make more granular decisions, based on conditions within conditions.

However, as highlighted in SICP, nested conditionals can make a program harder to read and debug.

As a best practice, we should strive to limit nesting and instead consider **refactoring** to improve readability.

Refactoring is the process of restructuring code to improve readability and maintainability without altering its behavior

```
age = 21
if age >= 18:
    if age >= 21:
        print("You can drink alcohol.")
    else:
        print("You are an adult, but not
allowed to drink yet.")
else:
    print("You are a minor.")
```

The return statement

The return statement, with or without a value, depending on whether the function is fruitful or void, allows us to terminate the execution of a function before (or when) we reach the end.

```
def print_square_root(x):  
    if x <= 0:  
        print("Positive numbers only,  
please.")  
    result = x**0.5  
    print("The square root of", x, "is",  
result)
```

Logical Operators (and, or, not)

To handle more complex conditions, we use logical operators like and, or, and not. Logical operators enable us to combine conditions in a meaningful way.

- **and**: Both conditions must be true.
- **or**: At least one condition must be true.
- **not**: Reverses the truth value of a condition.

```
temperature = 22
```

```
humidity = 70
```

```
if temperature > 20 and humidity > 60:  
    print("It's a warm and humid day.")
```


Logical Operators (and, or, not)

a	b	a or b
F	F	F
F	T	T
T	F	T
T	T	T

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

a	not a
F	T
T	F

Arithmetic Operations

The foundation of manipulating data

What are the basic operators?

Arithmetic Operations

The foundation of manipulating data

=

=

!=

Code Efficiency

Speed and Compute Capacity

As highlighted in *The Algorithm Design Manual*, efficient code is crucial to improving both speed and resource utilization. Algorithms should be designed to optimize performance, especially when working with large datasets or high computational demands.

Execution Time and **Memory Usage** are critical parameters of efficiency.

What is debugging

Bug origins & Error species

Origins of the “bug”

The term bug used to describe engineering errors dates back to at least 1889. It is said that it was popularized in computer science by Grace Hopper in 1947 when **an actual moth** was found causing issues in an early computer.

The process of finding and fixing bugs is known as debugging.



Debugging - Types of Errors

Syntax error

When the code structure violates the language rules. Python will stop execution and give you an error message.

```
if x > 10  
    print("Error")
```

What is the bug here?

Debugging - Types of Errors

Runtime Errors

When an error happens during the execution of the program. These errors are often due to unexpected conditions, such as dividing by zero or accessing a variable that hasn't been defined.

```
x = 10  
y = 0  
print(x / y)
```


Debugging - Types of Errors

Semantic Errors

When your program runs without crashing, but it doesn't produce the correct output. This type of error means the logic of your code is flawed.

```
x = 10  
y = 5  
print(x - y)
```

Debugging Techniques

What we can do about bugs?

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again.

Practice time!

Tutorial

Conditional Statements &
Debugging