Control structure: *Selections*

01204111 Computers and Programming

Chalermsak Chatdokmaiprai

Department of Computer Engineering
Kasetsart University

Outline

- **Boolean Type and Expressions**
- Fundamental Flow Controls
- Flowcharts: Graphical Representation of Controls
- Basic Selections: if statements
- Basic Selections: if-else statements
- Programming Examples

Python's Boolean Type: bool

- Type **bool** have two possible values: **True** and **False**

```
>>> print(True)
True
>>> print(False)
False
```

Values can be printed out.

```
>>> x = True
>>> y = False
>>> print(x)
True
>>> print(y)
False
```

Values can be assigned to variables.

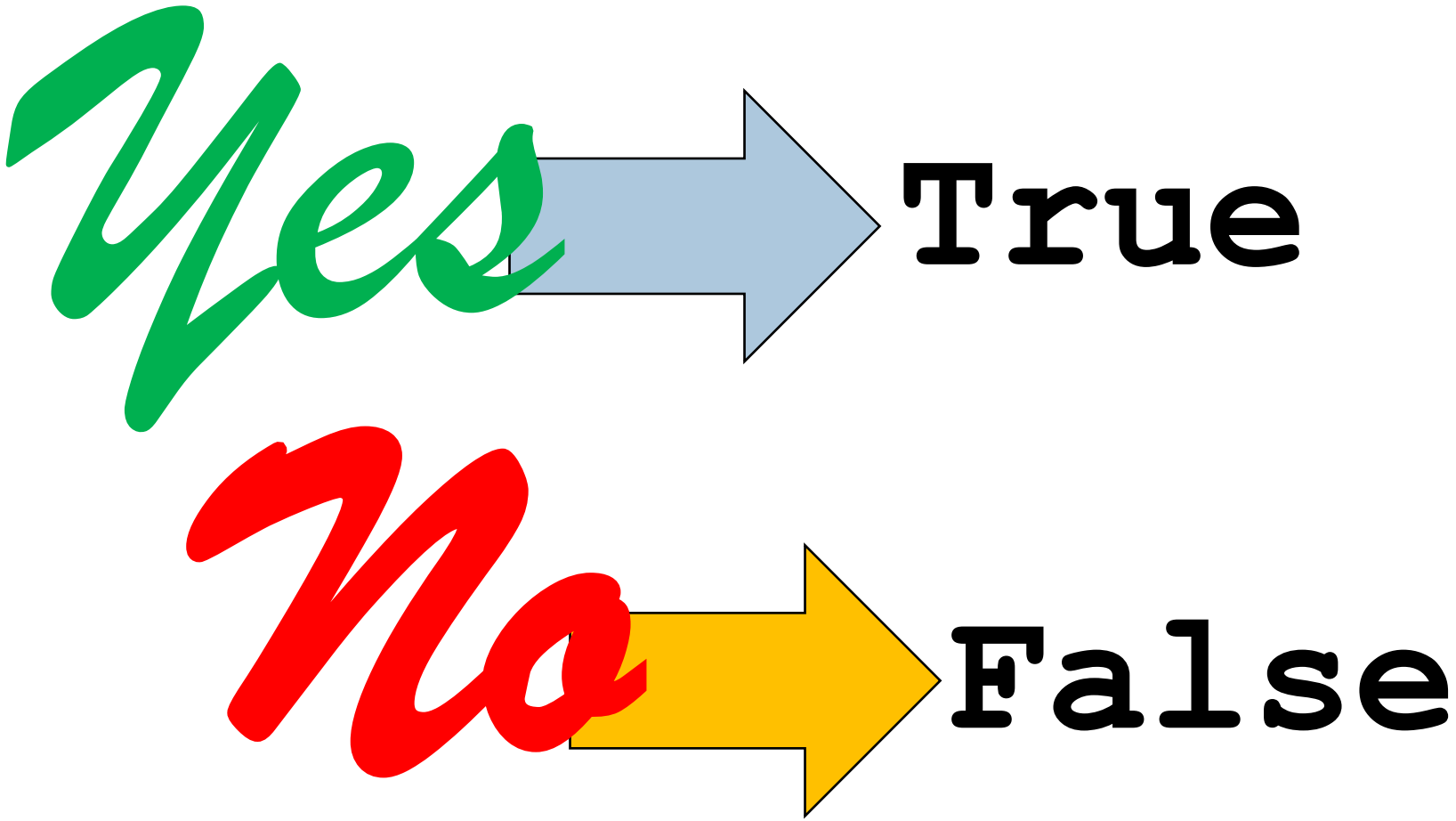
```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> type(x)
<class 'bool'>
```

You can check their types.

Boolean Expressions

- In Mathematics, a **Boolean expression** is an expression whose value is either **True** or **False**.
  - 20 > 10
  - 5 is a factor of 153
  - 18 is *not* a prime number **and** is divisible by 3
  - x > 5 **or** x < -5
- Evaluating a Boolean expression is just like answering a **yes/no question** in human languages:
  - Do you want the coffee? *(yes/no)*
  - Have you found the answer? *(yes/no)*
  - Is 20 greater than 10? *(yes/no)*
  - Is 5 a factor of 153? *(yes/no)*

Boolean Values



Boolean Expressions in Python

- In Python, a **Boolean expression** is **an expression of type bool**, which is evaluated to either **True** or **False**.

```
>>> print(5 > 3)
True
>>> print(5 < 3)
False
>>> print(5 > 3 and 'pig' != 'rat')
True
>>> x = 5
>>> pet = 'pig'
>>> print(x > 3 and pet != 'rat')
True
>>> print(x*2 > 100 or x+2 > 100)
False
```

You can use print() to evaluate a boolean expression and print the result.

In interactive mode, print() can be omitted.

```
>>> x > 3
True
>>> x > 10 or x < 0
False
```

Watch Out!

- Python is case-sensitive so …
  - `False` and `false` are ***not*** the same.
- Python's bool constants are written precisely as:
  - `True`, or
  - `False`

How to write a Boolean expression in Python

- We can use a **relational operator** to compare two things:

Meaning	Operator
Equal	==
Not equal	!=
Greater than	>
Greater than or equal	>=
Less than	<
Less than or equal	<=

How to write a Boolean expression in Python

- We can use **logical operators** to combine two or more Boolean expressions:

Meaning	Operator
Boolean AND	**and**
Boolean OR	**or**
Boolean NOT	**not**

Quick Review

p	q	p and q
True	True	True
True	False	False
False	True	False
False	False	False

p	not p
True	False
False	True

p	q	p or q
True	True	True
True	False	True
False	True	True
False	False	False

George Boole, 1815-1864
An English mathematician
The founder of Boolean Algebra

[Image via http://www.storyofmathematics.com/19th_boole.html]

Hands-On Examples

```
>>> i = 10
>>> j = 15
>>> print(j < i)
False
```

A boolean expression can be assigned to a variable.

```
>>> r = i+2 >= 10
>>> print(r)
True
```

```
>>> print((i%2) != 0)
False
>>> print(not ((i%2) == 0))
False
```

Both expressions are logically equivalent.

```
>>> print(i+j >= 5 and i+j <= 25)
True
>>> print(5 <= i+j <= 25)
True
```

Both expressions are logically equivalent.

```
>>> print((not r) or (i > 20 and i <= j))
False
```

You can nest them if you know what you mean.

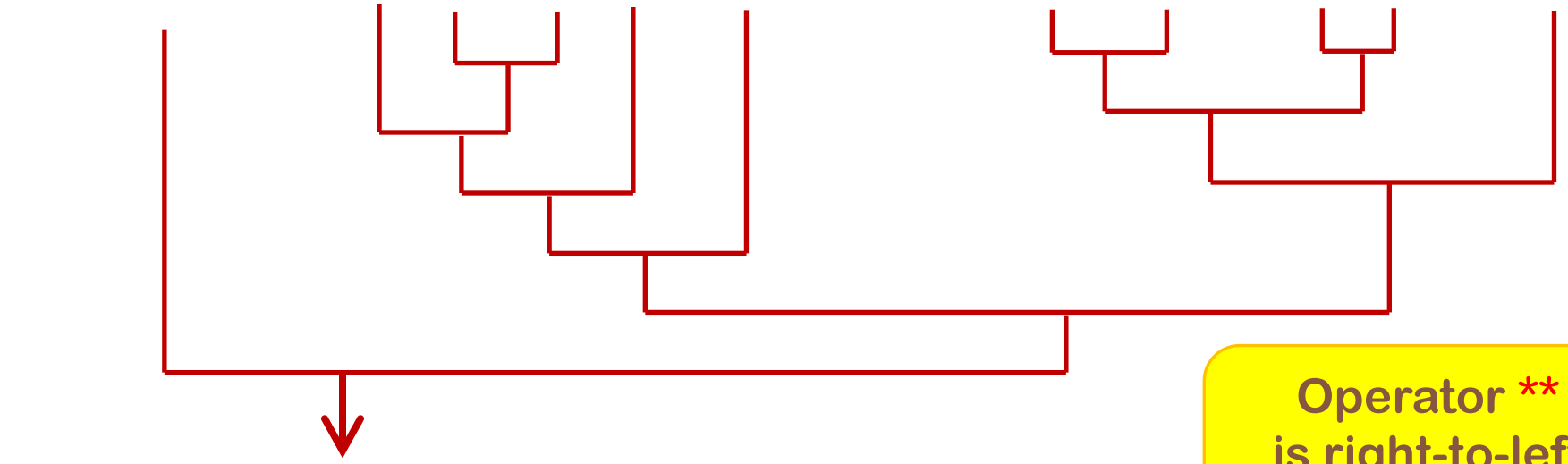
Python Operator Precedence

- From the highest precedence to the lowest down the table.
- Operators on the same row have the same precedence.

Category	Operators	Associativity
Subscription, call, attribute	`a[x]  f(x)  x.attribute`	left to right
Exponentiation	`**`	**right to left**
Unary sign	`+x  -x`	left to right
Multiplicative	`*  /  //  %`	left to right
Additive	`+  -`	left to right
Relational (comparison)	`==  !=  <  >  <=  >=`	left to right
Boolean NOT	`not`	left to right
Boolean AND	`and`	left to right
Boolean OR	`or`	left to right

Operator Precedence: Examples

```
passed = i/j**3-2<10 or math.sqrt(i*j)>=20
```



The result is the value assigned to the variable passed

Operator ** is right-to-left associative.

```
>>> 4**2**3
65536
>>> (4**2)**3
4096
>>> 4**(2**3)
65536
```

More Example

- Write a function to check if a given number is a root of the equation

$$x^2 + 3x - 10 = 0$$

```
>>> def isroot(x):
        return x**2 + 3*x - 10 == 0
```

Define a function to do the task.

```
>>> print(isroot(2))
True
>>> isroot(-3)
False
>>> isroot(-5)
True
>>> isroot(0)
False
```

Call the function to check if the given number is a root.

In interactive mode, print() can be omitted.

Outline

- Boolean Data Type and Expressions
- **Fundamental Flow Controls**
- Flowcharts: Graphical Representation of Controls
- Basic Selections: if statements
- Basic Selections: if-else statements
- Programming Examples

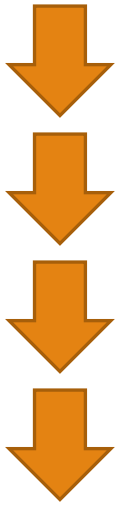
Fundamental Flow Controls

- **Sequence**
- **Subroutine**

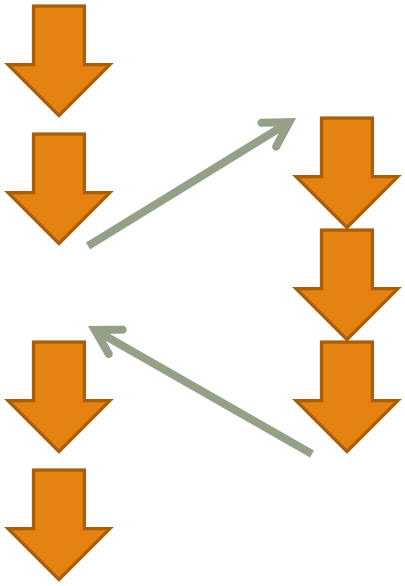
You have already learned and used these two control structures.

- **Selection** (or **Branching**)
- **Repetition** (or **Iteration** or **Loop**)

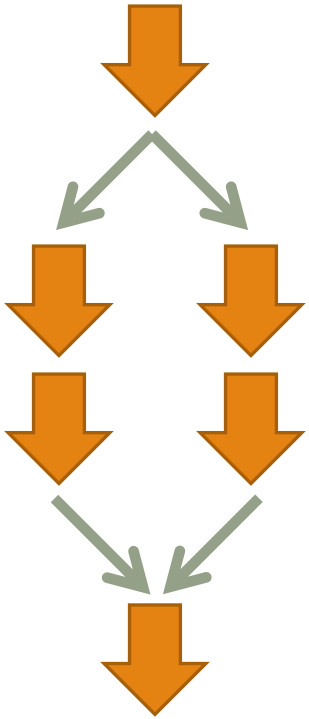
Schematic View of Flow Controls



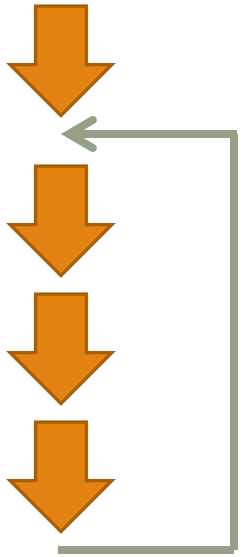
Sequence



Subroutine



Selection



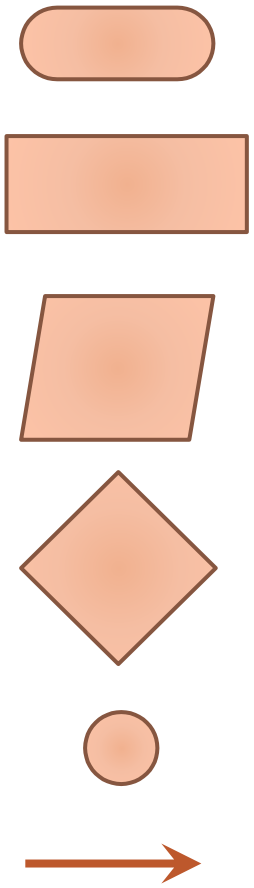
Repetition

Outline

- Boolean Data Type and Expressions
- Fundamental Flow Controls
- **Flowcharts: Graphical Representation of Controls**
- Basic Selections: if statements
- Basic Selections: if-else statements
- Programming Examples

Flowcharts: Graphical Representation of Controls

Basic flowchart symbols:



Terminator

Process

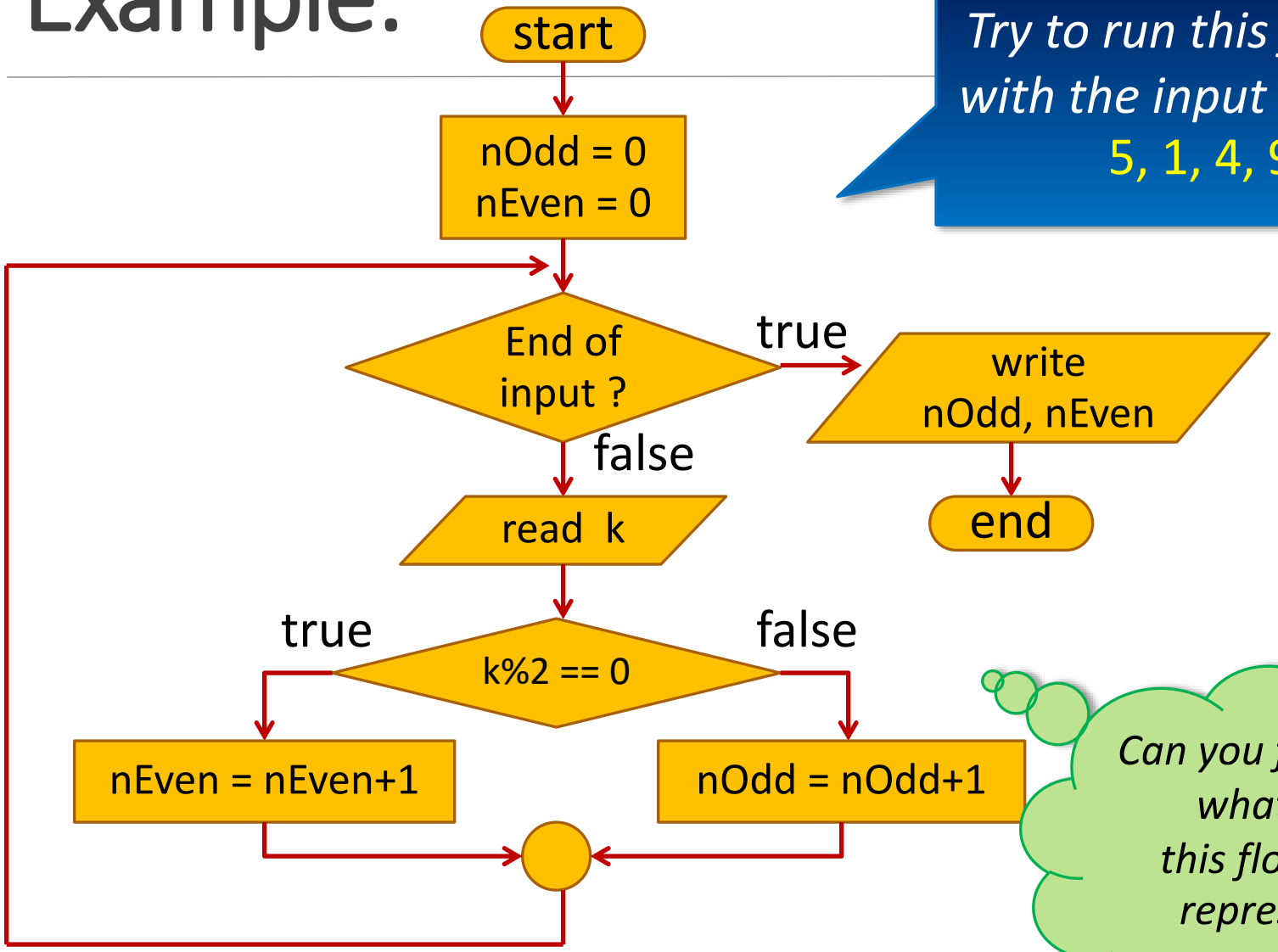
Input/output

Condition

Connector

Flow line

Example:



Try to run this flowchart with the input sequence:
5, 1, 4, 9, 8

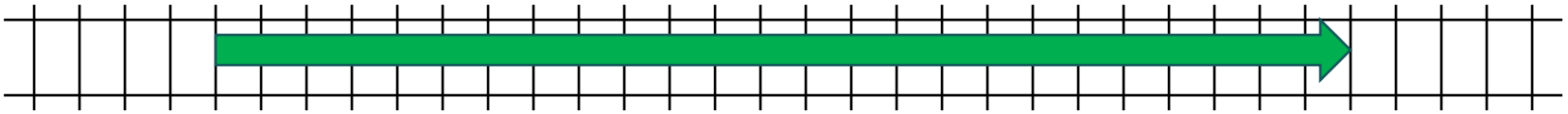
Can you figure out what task this flowchart represents?

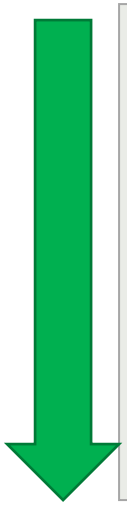
Outline

- Boolean Data Type and Expressions
- Fundamental Flow Controls
- Flowcharts: Graphical Representation of Controls
- **Basic Selections: if statements**
- Basic Selections: if-else statements
- Programming Examples

Normal Sequential Flow

- This is the *default* program flow unless specified otherwise.



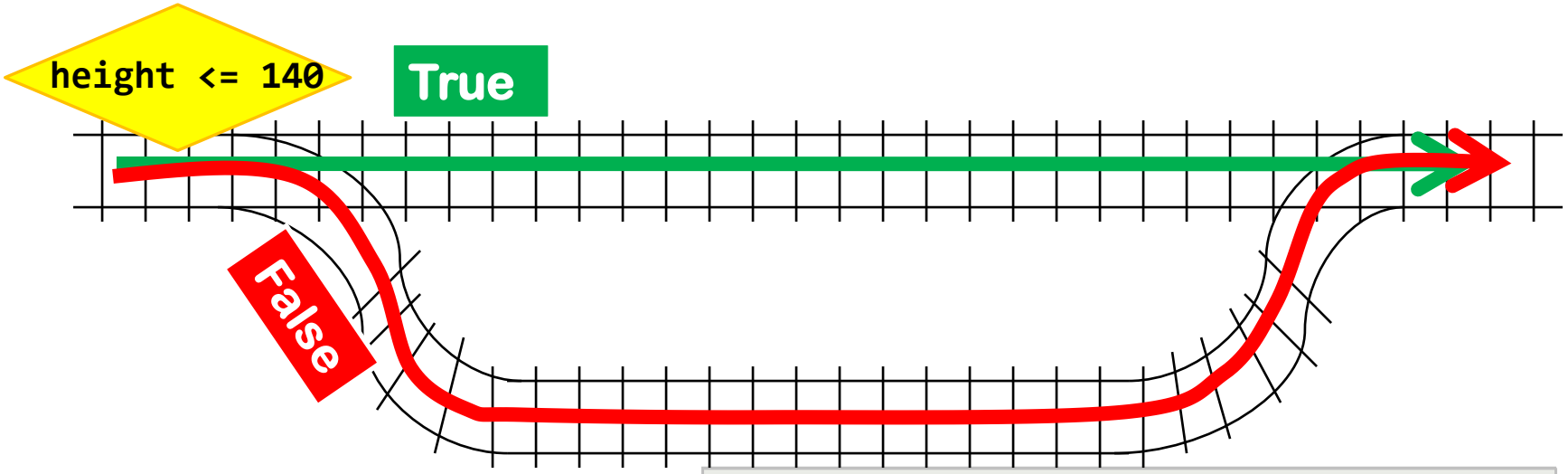


```
x = int(input())
y = int(input())
print(x+y)
print("Hello",x)
z = x * y + 10
print(z)
```

[Images reproduced by kind permission of Chaiporn Jaikaeo & Jittat Fakcharoenphol]

Selection flow with if-statement

- Also called conditional execution



When height is 160

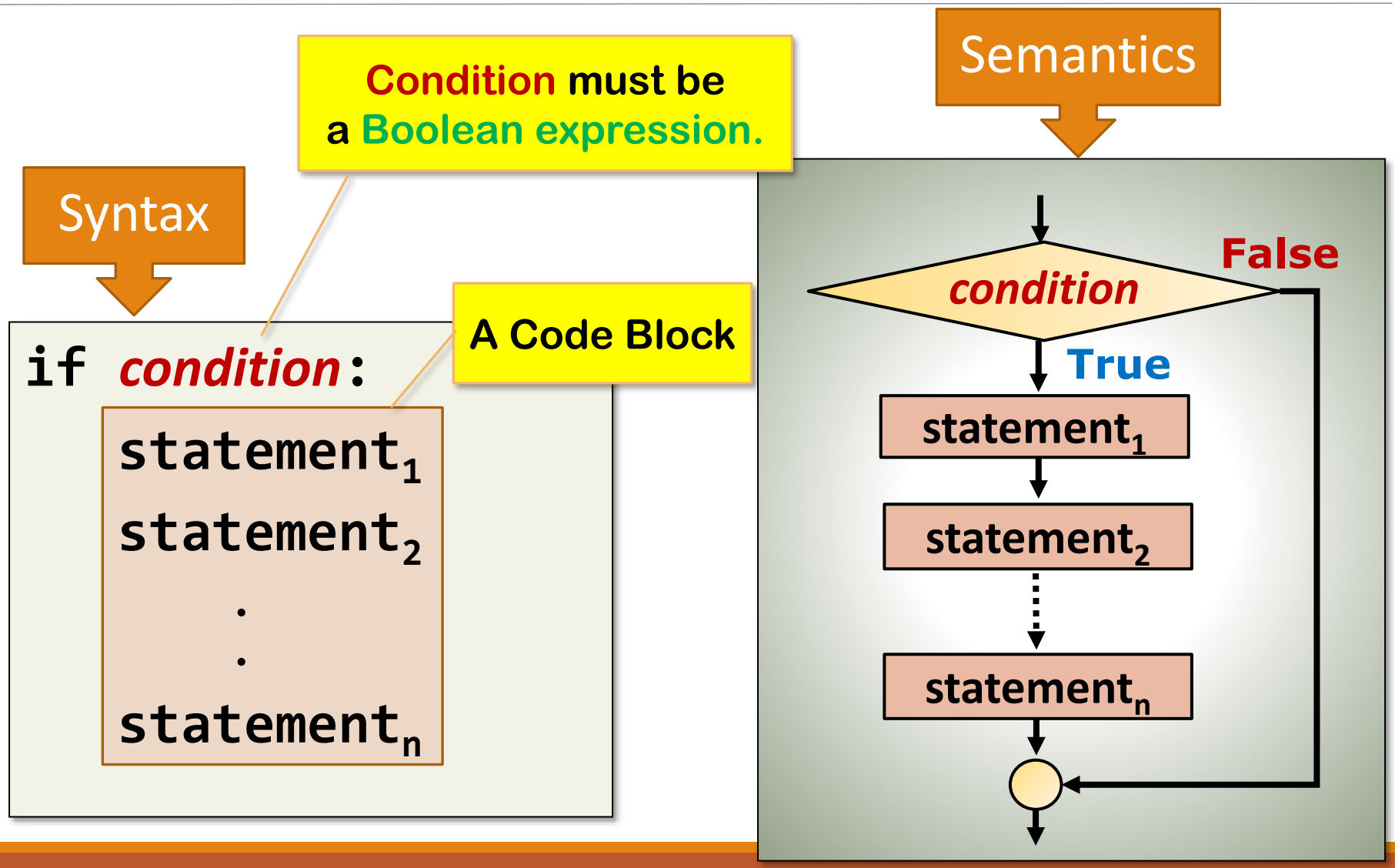
```
price = 40
if height <= 140:
    print('Hello kids!')
    price = 0
print('price =', price)
```

Basic Selection: if statement

- **`if`** statement is used to decide whether a **code block** is to be executed or not, depending on a **condition**.

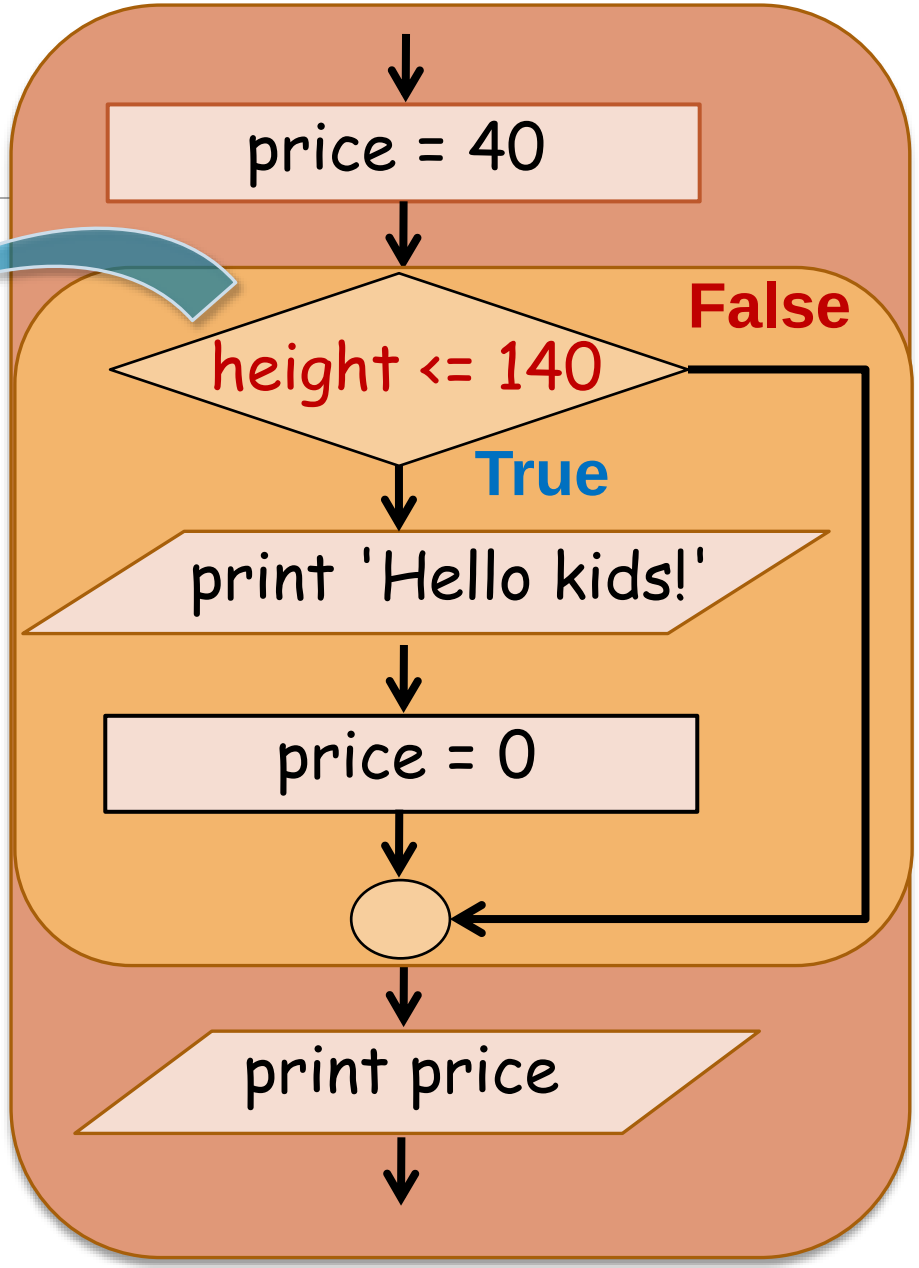
- The statements in the code block will be executed only if the condition is **True**.

Basic Selection: if statement



Example


```
price = 40
if height <= 140:
    print('Hello kids!')
    price = 0
print('price =', price)
```

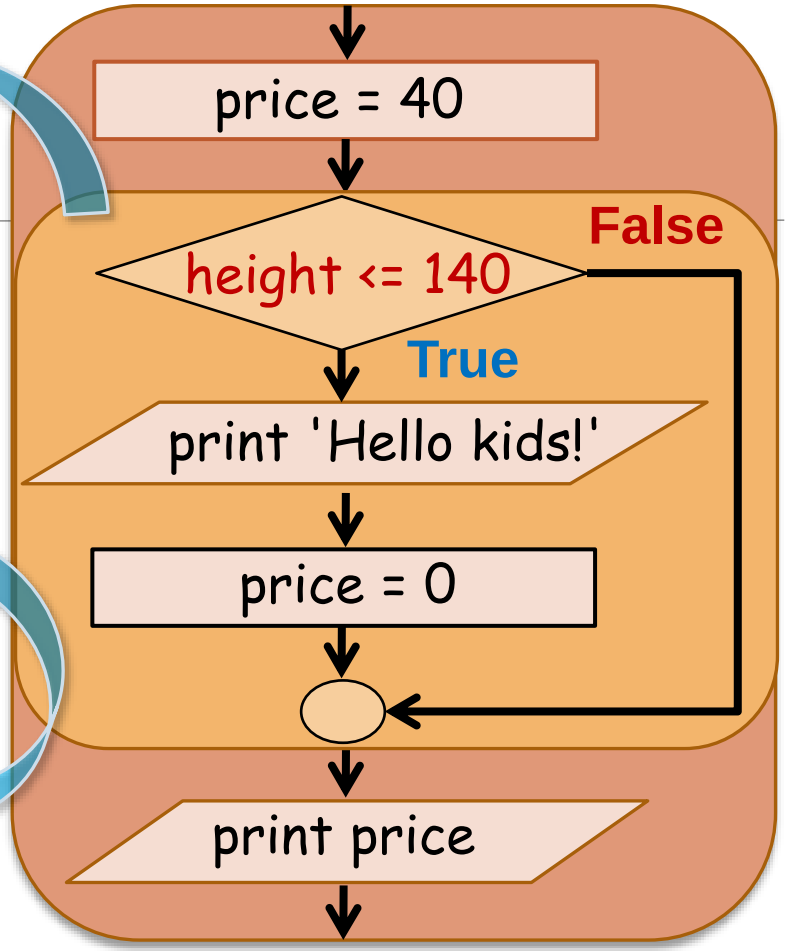


[Image: courtesy of Mass Rapid Transit Authority of Thailand]

Make it a function

```
price = 40
if height <= 140:
    print('Hello kids!')
    price = 0
print('price =', price)
```

```
def mrt_fee(height):
    price = 40
    if height <= 140:
        print('Hello kids!')
        price = 0
    print('price =', price)
```

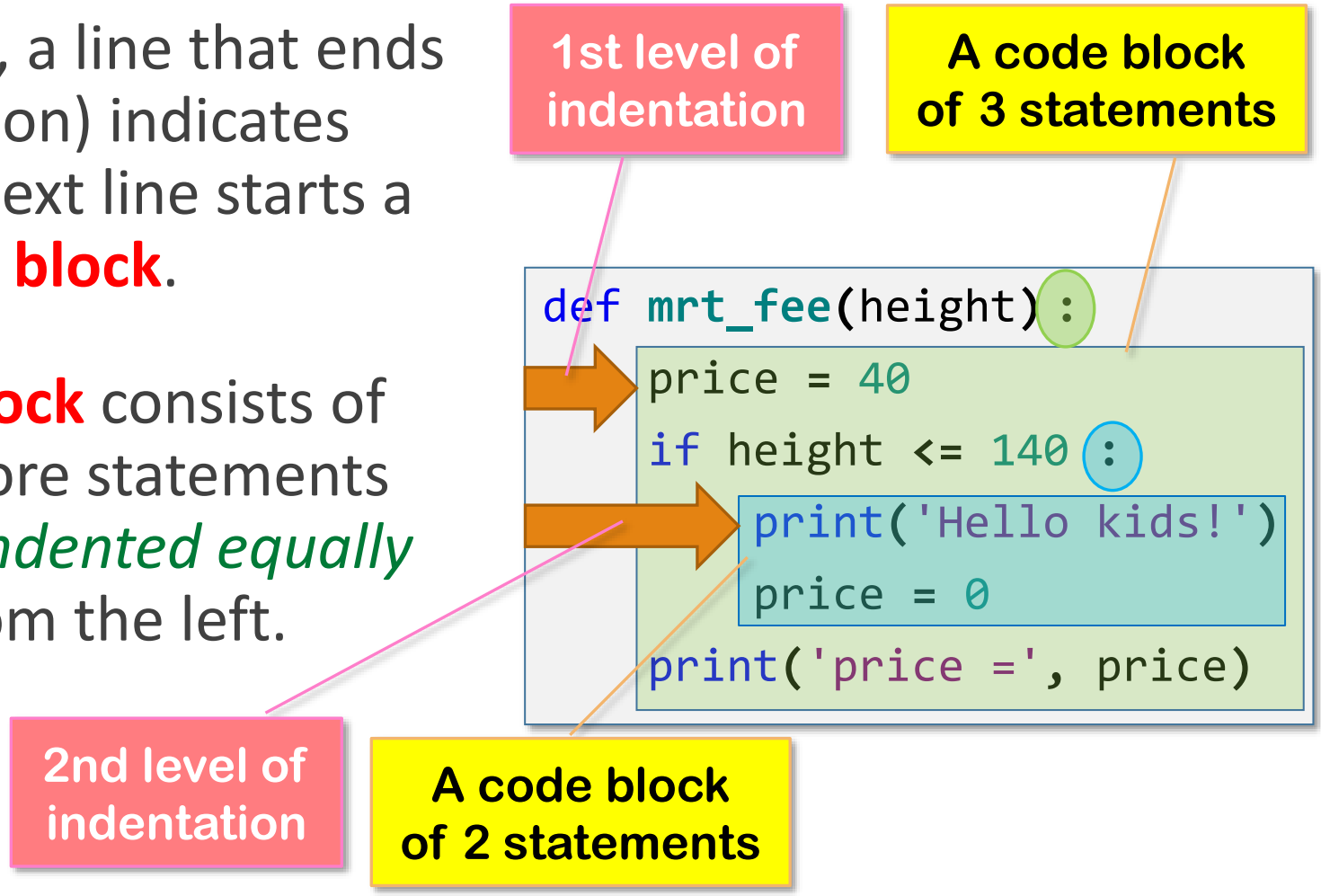


Test it.

```
>>> mrt_fee(140)
Hello kids!
price = 0
>>> mrt_fee(150)
price = 40
>>>
```

Code Blocks

- In Python, a line that ends with **:** (colon) indicates that the next line starts a new **code block**.
- A **code block** consists of one or more statements that are *indented equally deeply* from the left.



Be Careful

- Python uses the concept of blocks extensively.
- Thus, you must be *very* careful about indentation.

[Images reproduced by kind permission of Chaiporn Jaikaeo & Jittat Fakcharoenphol]

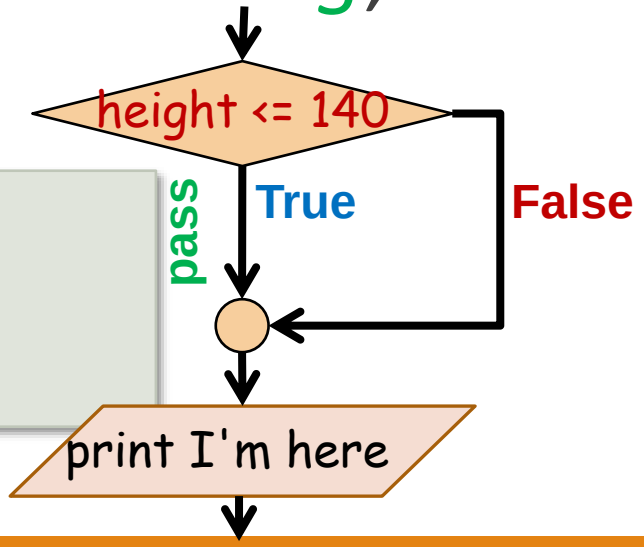
`pass` statement for an empty block

- In Python, we *cannot* have an empty block.

```
if height <= 140:
print("I'm here")
```

- If you want a block that *does nothing*, use the pass statement.

```
if height <= 140:
    pass
print("I'm here")
```



More Example: Find the larger of two integers

The function max_of_two()

- receives two number parameters a and b.
- returns the larger of them.

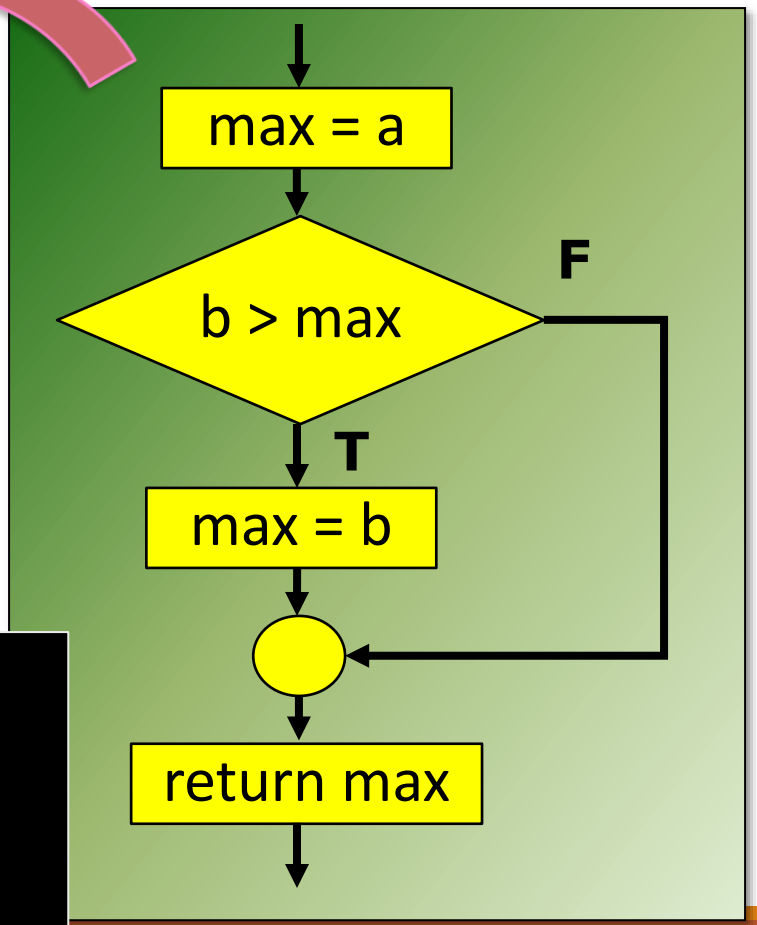
Python Code

```
def max_of_two(a, b):
    max = a
    if b > max:
        max = b
    return max
```

Test it.

```
>>> max_of_two(2, 3)
3
>>> max_of_two(3, 2)
3
>>> max_of_two(3, 3)
3
```

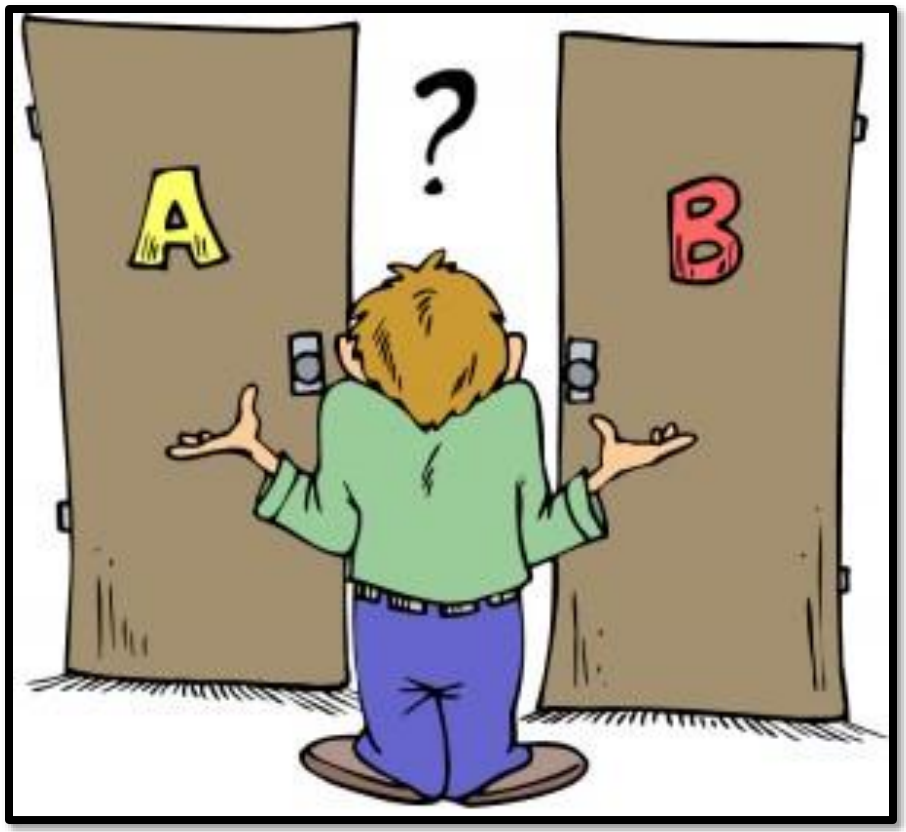
Flow of execution



Outline

- Boolean Data Type and Expressions
- Fundamental Flow Controls
- Flowcharts: Graphical Representation of Controls
- Basic Selections: if statements
- **Basic Selections: if-else statements**
- Programming Examples

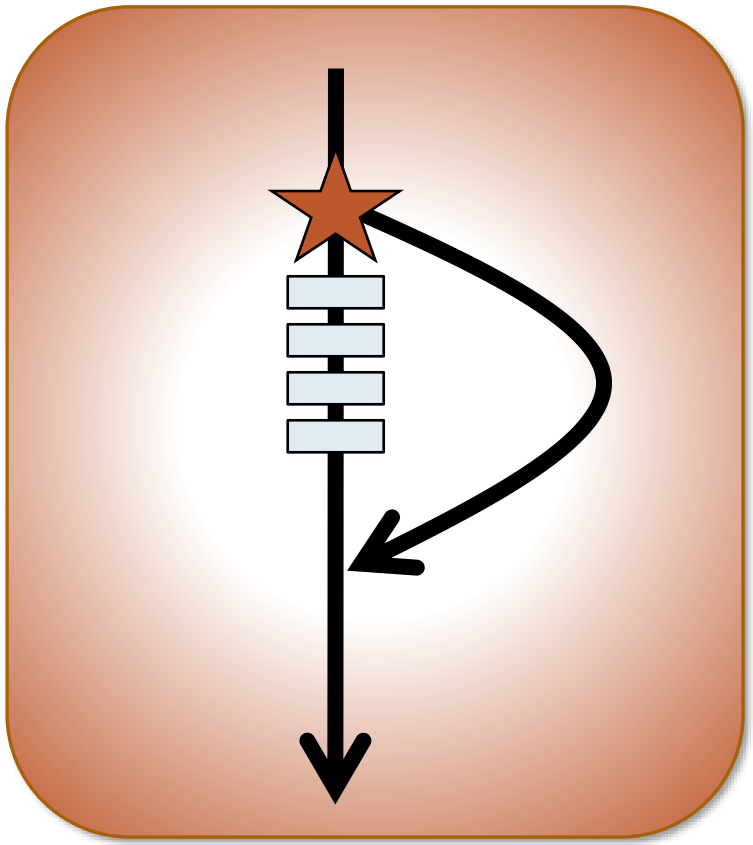
if-else statements : Alternative Execution



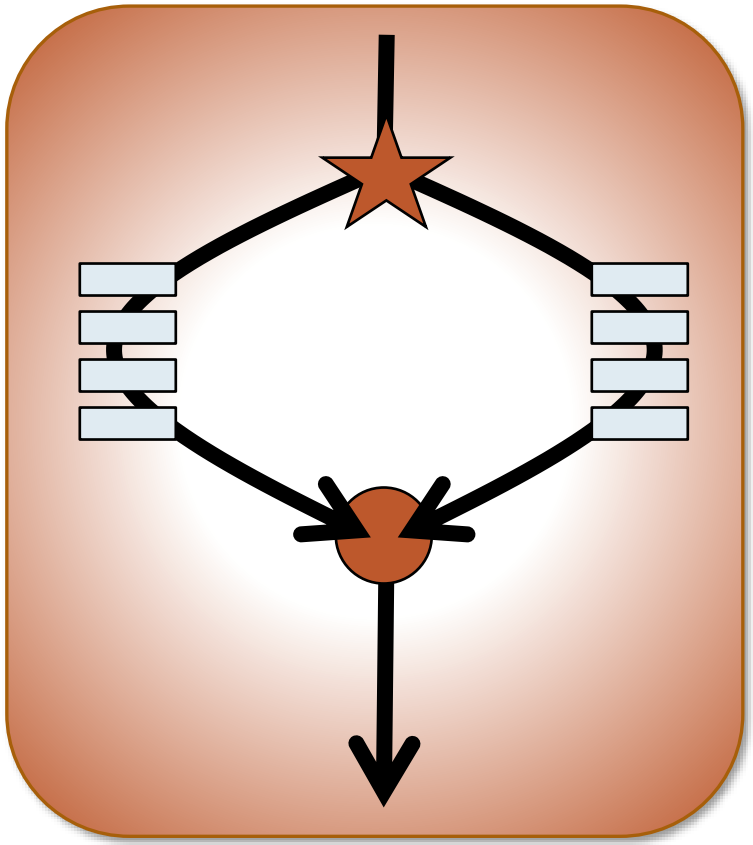
Source: http://splinedoctors.com/2009/02/hurry-up-and-choose/

if versus if-else

if statement



if-else statement



[Images reproduced by kind permission of Chaiporn Jaikaeo & Jittat Fakcharoenphol]

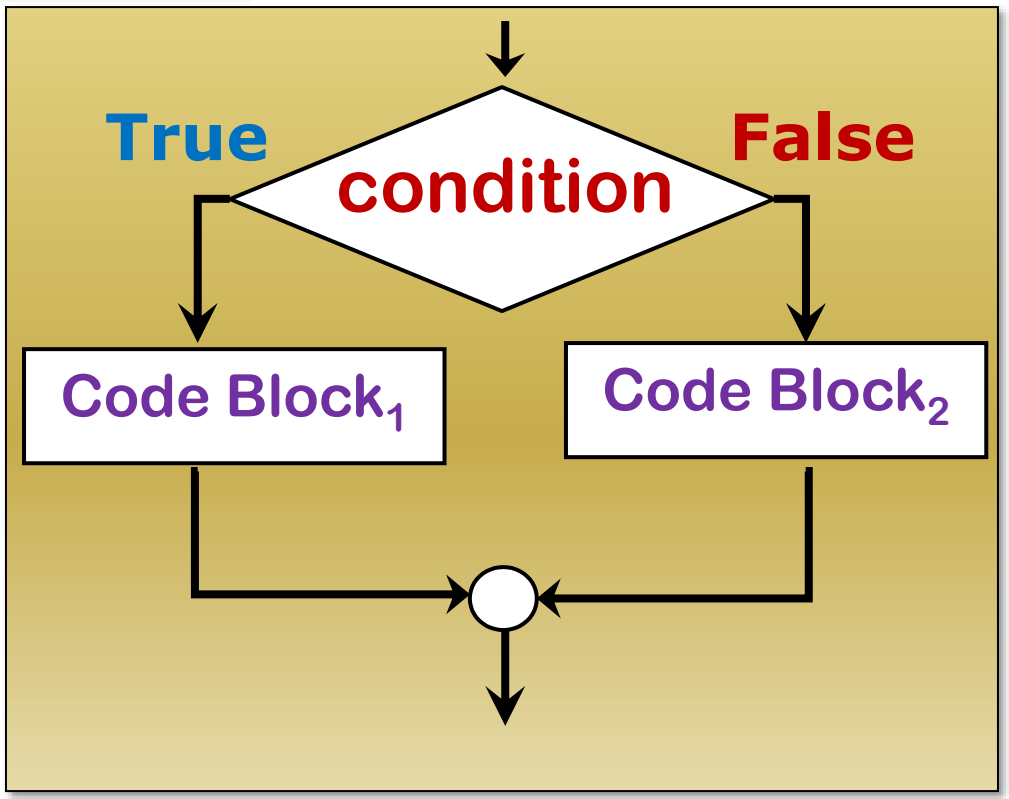
Alternative Execution: if-else statement

Python Syntax

Condition is a Boolean expression.

Semantics

```
if   condition:
        Code Block1
else:
        Code Block2
```



Don't forget the colons and indentation

Example: The function max_of_two() revisited

Python Code

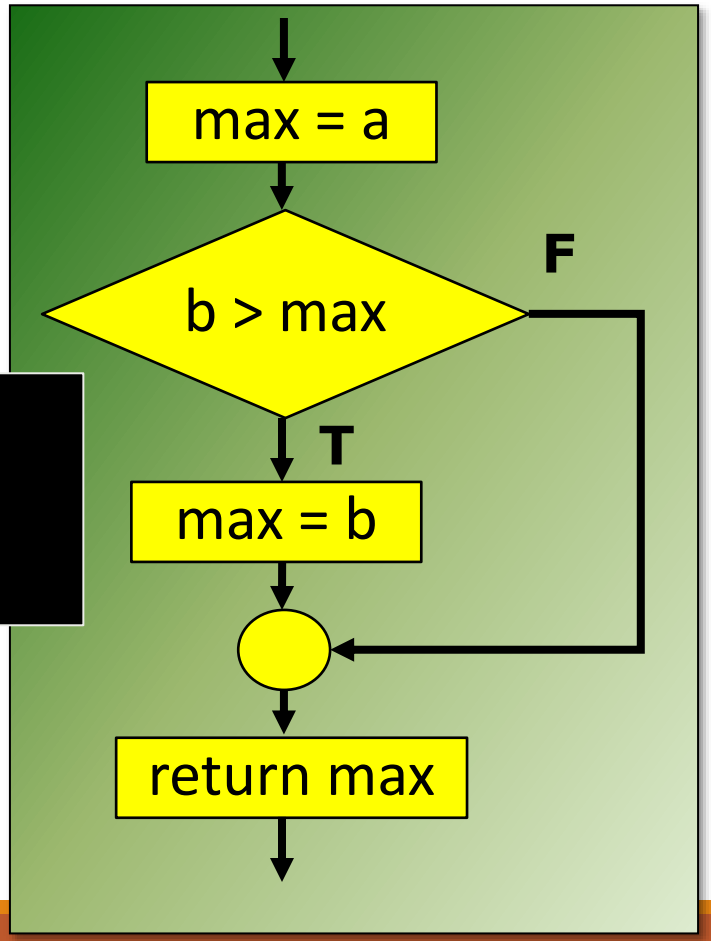
```
def max_of_two(a, b):
    max = a
    if b > max:
        max = b
    return max
```

```
>>> max_of_two(2, 3)
3
>>> max_of_two(3, 3)
3
```

This version:

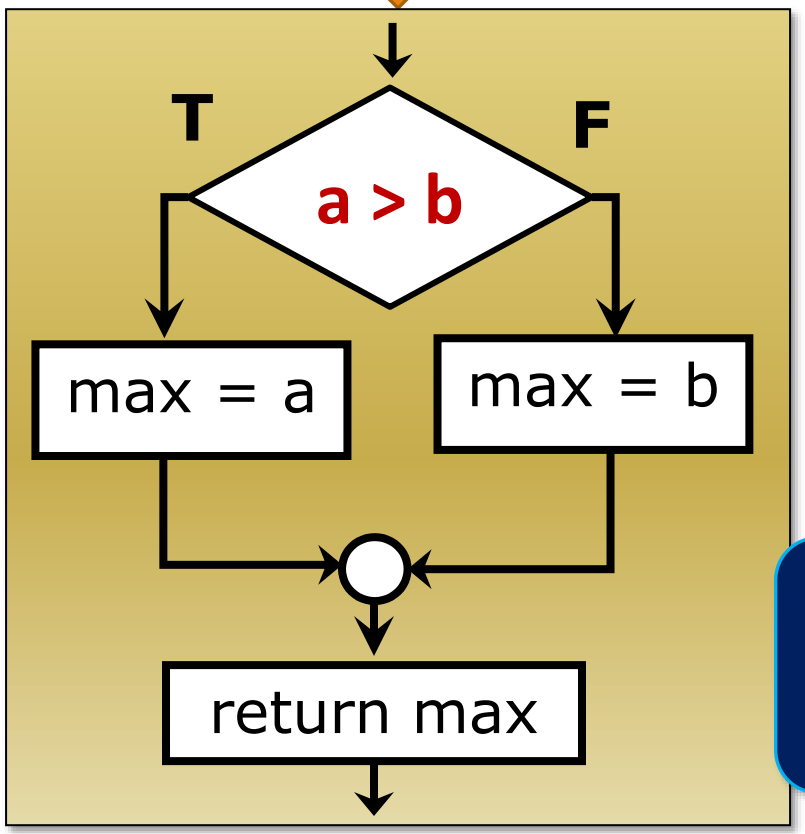
- *uses **if** (without **else**) statement*
- *executes **one or two** assignments*

Flow of execution



Example: another way to write max_of_two()

Flow of execution



Python Code

```
def max_of_two(a, b):
    if a > b:
        max = a
    else:
        max = b
    return max
```

This version:

- *uses if-else statement*
- *always executes only one assignment*

Or a slimmer version!

```
def max_of_two(a, b):
    if a > b:
        return a
    else:
        return b
```

Test it.

```
>>> print(max_of_two(3,2))
3
>>> max_of_two(3, 2)
3
>>> max_of_two(2, 3)
3
>>> x = 5
>>> max_of_two(3*x, x**2)
25
```

In interactive mode, print() can be omitted.

Example:
if and else code blocks with several statements

```
def payment(nitems, itemprice):
    price = nitems*itemprice
    if nitems > 10:
        print('You got 10% discount.')
        price = 0.9*price
        print(f'You also got {nitems//3} stamps.')
    else:
        print('You got 5% discount.')
        price = 0.95*price
    print(f'Total payment is {price} bahts.')
```

```
>>> payment(10, 3)
You got 5% discount.
Total payment is 28.5 bahts.
>>> payment(itemprice=5, nitems=20)
You got 10% discount.
You also got 6 stamps.
Total payment is 90.0 bahts.
>>>
```

Test it.

code blocks inside the if-else statement.

Outline

- Boolean Data Type and Expressions
- Fundamental Flow Controls
- Flowcharts: Graphical Representation of Controls
- Basic Selections: if statements
- Basic Selections: if-else statements
- **Programming Examples**

Task: Solving quadratic equations

- Given the three coefficients a, b, and c of a quadratic equation $ax^2 + bx + c = 0$ where $a \neq 0$, find the **roots** of the equation.

A ***root*** is a value of x that satisfies the equation

Solving quadratic equations - I/O Specification

Sample Run

```
Enter 1st coefficient: 0
Enter 2nd coefficient: -2
Enter 3rd coefficient: 5
1st coefficient can't be zero. Program exits.
```

Sample Run

```
Enter 1st coefficient: 1
Enter 2nd coefficient: 8
Enter 3rd coefficient: 16
Only one real root: -4
```

Sample Run

```
Enter 1st coefficient: 2
Enter 2nd coefficient: -1
Enter 3rd coefficient: -1
Two real roots: 1 and -0.5
```

Sample Run

```
Enter 1st coefficient: 5
Enter 2nd coefficient: 2
Enter 3rd coefficient: 1
Two complex roots: -0.2+0.4i and -0.2-0.4i
```

Solving quadratic equations - Ideas

- The *roots* of a quadratic equation $ax^2 + bx + c = 0$ can be calculated by the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- The term ***$b^2 - 4ac$*** in the formula is called the **discriminant** (D) of the equation because it can discriminate between the possible types of roots.

Solving quadratic equations - Ideas

The discriminant $D = b^2 - 4ac$ of the equation determines the type of roots as follows:

- *If D > 0, there are two real roots:* $\frac{-b+\sqrt{D}}{2a}$ *and* $\frac{-b-\sqrt{D}}{2a}$
- *If D = 0, there is only one real root:* $\frac{-b}{2a}$
- *If D < 0, there are two complex roots:*

$$\frac{-b}{2a}+i\frac{\sqrt{-D}}{2a} \quad \text{and} \quad \frac{-b}{2a}-i\frac{\sqrt{-D}}{2a}$$

Now we've got enough information to write the program.

Next: Developing the program

We are going to demonstrate a useful, effective development technique called

Incremental Development

together with

Incremental test

Topmost Steps

- **The main routine:**
  1. reads the three coefficients *a*, *b*, and *c*, making sure that *a* is not zero.
  2. uses *a*, *b*, and *c* to solve and output the roots.

```
import sys
from math import sqrt

# ----- main ----- #
a, b, c = read_coefficients()
if a == 0:
    print("1st coefficient can't be zero. Program exits.")
    sys.exit()   # can't do anything more with bad input
solve_and_output(a, b, c)
```

*The supreme commander **main** usually doesn't do things himself. He only gives orders.*

exit this running program immediately

Before going on, we'd better test it

We add some **scaffolding code** to be able to test the main routine.

```
import sys
from math import sqrt

def read_coefficients():
    print('In read_coefficients:')   # dummy code
    return 1, 2, 3                   # dummy code

def solve_and_output(a, b, c):
    print("In solve_and_output:", a, b, c)  # dummy code

# ----- main ----- #
a, b, c = read_coefficients()
print('In main: main receives', a, b, c)    # dummy code
if a == 0:
    print("1st coefficient can't be zero. Program exits.")
    sys.exit()   # can't do anything more with bad input
solve_and_output(a, b, c)
```

Test Results

Test run

```
In read_coefficients:
In main: main receives 1 2 3
In solve_and_output: 1 2 3
```

```
def read_coefficients():
    print('In read_coefficients:')    # dummy code
    return 0, 2, 3                    # dummy code
```

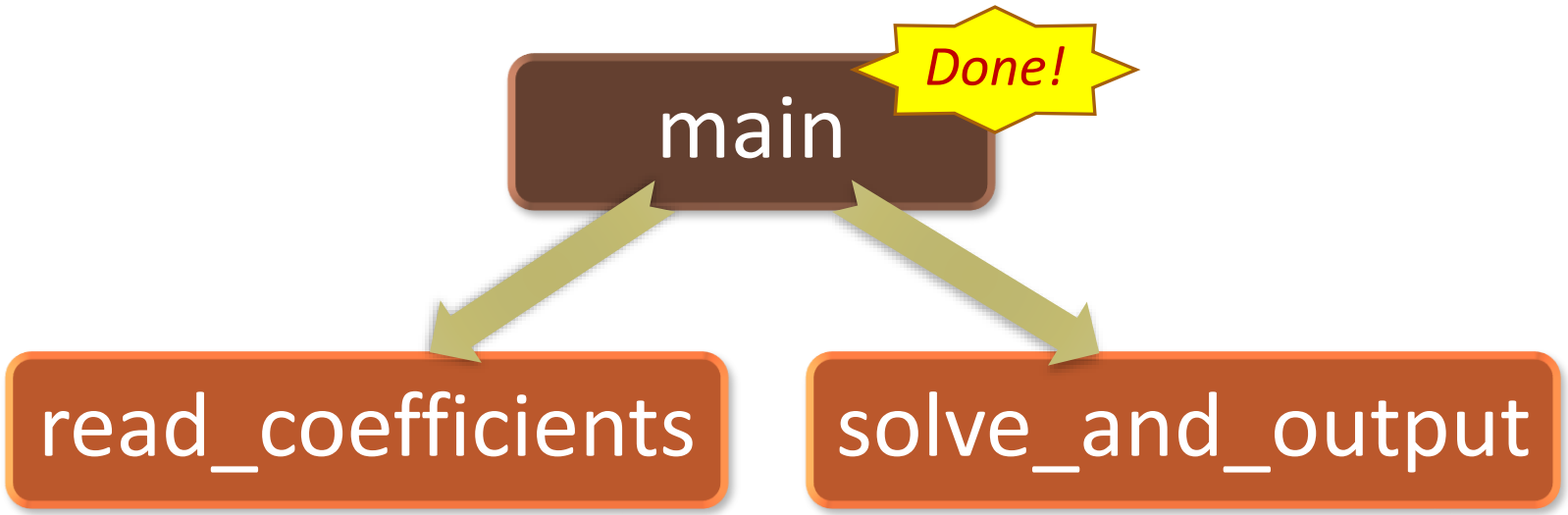
Change to 0 and rerun it

Test run

```
In read_coefficients:
In main: main receives 0 2 3
1st coefficient can't be zero. Program exits.
```

Main routine has been tested and it works correctly.

What we've done so far



This schema is called
the subroutine call tree.

Next: Reading the inputs

- The function read_coefficients()
  reads and returns the coefficients *a*, *b*, and *c*.

```
def read_coefficients():
    a = float(input('Enter 1st coefficient: '))
    b = float(input('Enter 2nd coefficient: '))
    c = float(input('Enter 3rd coefficient: '))
    return a, b, c
```

Fit it in, then test the program again

Put new code here.

```
import sys
from math import sqrt

def read_coefficients():
    a = float(input('Enter 1st coefficient: '))
    b = float(input('Enter 2nd coefficient: '))
    c = float(input('Enter 3rd coefficient: '))
    return a, b, c

def solve_and_output(a, b, c):
    print("In solve_and_output:", a, b, c)  # dummy code

# ----- main ----- #
a, b, c = read_coefficients()
print('In main: main receives', a, b, c)    # dummy code
if a == 0:
    print("1st coefficient can't be zero. Program exits.")
    sys.exit()   # can't do anything more with bad input
solve_and_output(a, b, c)
```

Test Results

Test run

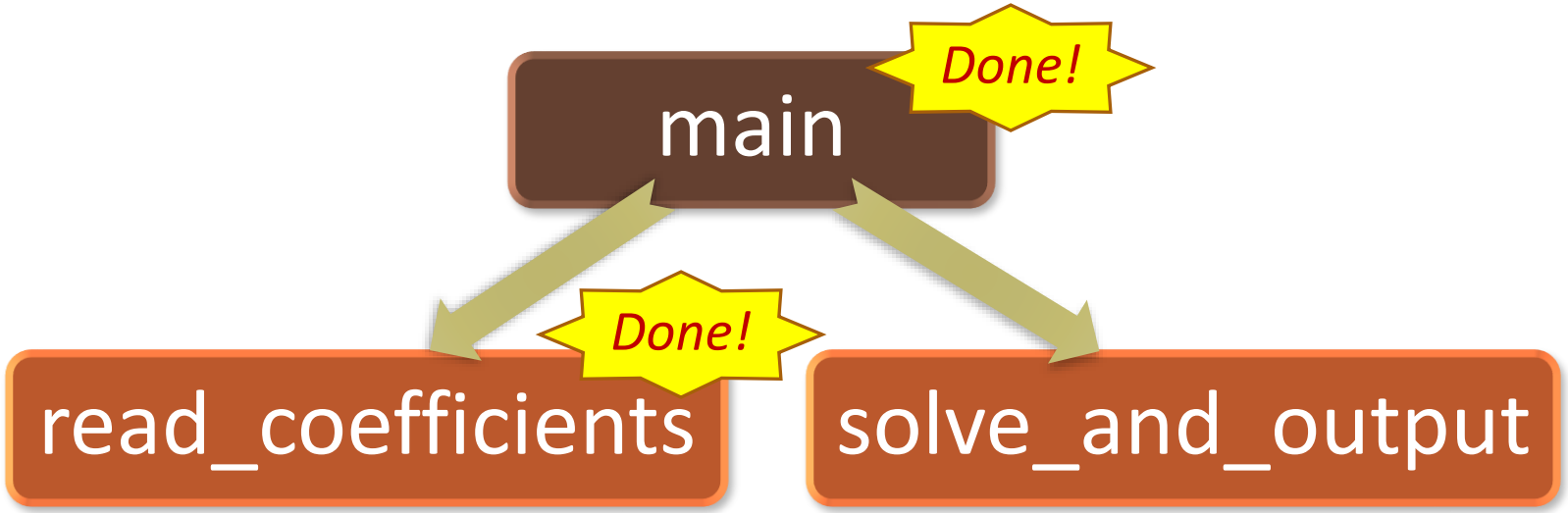
```
Enter 1st coefficient: 1
Enter 2nd coefficient: 2
Enter 3rd coefficient: 3
In main: main receives 1.0 2.0 3.0
In solve_and_output: 1.0 2.0 3.0
```

Test run

```
Enter 1st coefficient: 0
Enter 2nd coefficient: 1
Enter 3rd coefficient: 2
In main: main receives 0.0 1.0 2.0
1st coefficient can't be zero. Program exits.
```

So far so good !

What we've done so far



Next: The Solving Engine

- The function solve_and_output()
  1. computes the **discriminant**.
  2. uses the discriminant to select either the function to find **real roots** or the one to find **complex roots**.

```
def solve_and_output(a, b, c):
    disc = b*b - 4*a*c
    if disc >= 0: # has real roots
        compute_real_roots(a, b, disc)
    else:         # has complex roots
        compute_complex_roots(a, b, disc)
```

Formula for roots needs discriminant rather than c.

Formula for roots needs discriminant rather than c.

Fit it in, then test the program again

```
import sys
from math import sqrt
def read_coefficients():
    a = float(input('Enter 1st coefficient: '))
    b = float(input('Enter 2nd coefficient: '))
    c = float(input('Enter 3rd coefficient: '))
    return a, b, c
```

Put new code here.

```
def solve_and_output(a, b, c):
    disc = b*b - 4*a*c
    if disc >= 0:   # has real roots
        compute_real_roots(a, b, disc)
    else:           # has complex roots
        compute_complex_roots(a, b, disc)
```

And some new scaffolds for testing

```
def compute_real_roots(a, b, disc):
    print("In compute_real_roots:", a, b, disc)  # dummy code
def compute_complex_roots(a, b, disc):
    print("In compute_complex_roots:", a, b, disc) # dummy code
```

Why can we put these two functions after `solve_and_output()` that calls them?

```
# ----- main ----- #
a, b, c = read_coefficients()
if a == 0:
    print("1st coefficient can't be zero. Program exits.")
    sys.exit()   # can't do anything more with bad input
solve_and_output(a, b, c)
```

Test Results

Test run

```
Enter 1st coefficient: 1
Enter 2nd coefficient: -4
Enter 3rd coefficient: 4
In compute_real_roots: 1.0 -4.0 0.0
```

discriminant is 0 (real root)

Test run

```
Enter 1st coefficient: 2
Enter 2nd coefficient: 1
Enter 3rd coefficient: -1
In compute_real_roots: 2.0 1.0 9.0
```

discriminant > 0 (real root)

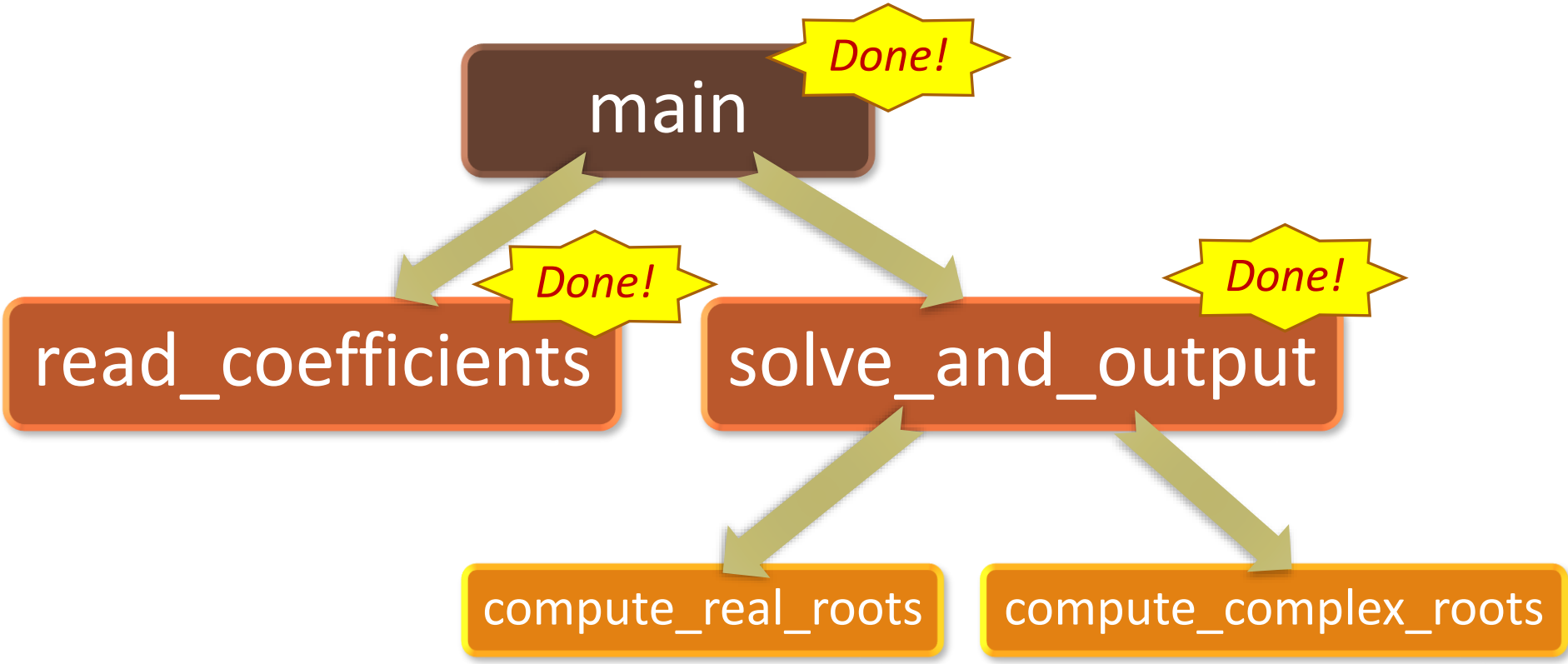
Test run

```
Enter 1st coefficient: 2
Enter 2nd coefficient: 1
Enter 3rd coefficient: 1
In compute_complex_roots: 2.0 1.0 -7.0
```

discriminant < 0 (complex root)

So far so smooth !

What we've done so far



Before going further, let's recall the formula:

The discriminant $D = b^2 - 4ac$ of the equation determines the type of roots as follows:

- *If D > 0, there are two real roots:* $\frac{-b+\sqrt{D}}{2a}$ *and* $\frac{-b-\sqrt{D}}{2a}$
- *If D = 0, there is only one real root:* $\frac{-b}{2a}$
- *If D < 0, there are two complex roots:*

$$\frac{-b}{2a}+i\frac{\sqrt{-D}}{2a} \quad \text{and} \quad \frac{-b}{2a}-i\frac{\sqrt{-D}}{2a}$$

Next: Compute the real roots

- The function compute_real_roots()
  1. uses the *discriminant* to select either the formula for ***one*** real root or ***two*** real roots.
  2. computes and outputs the root(s).

```
def compute_real_roots(a, b, disc):
    if disc == 0:
        root = -b / (2*a)
        print(f'Only one real root: {root}')
    else:
        root1 = (-b + sqrt(disc)) / (2*a)
        root2 = (-b - sqrt(disc)) / (2*a)
        print(f'Two real roots: {root1} and {root2}')
```

$\frac{-b}{2a}$

$\frac{-b+\sqrt{D}}{2a}$

$\frac{-b-\sqrt{D}}{2a}$

Fit it in, then test the program again

```
import sys
from math import sqrt
def read_coefficients():
    a = float(input('Enter 1st coefficient: '))
    b = float(input('Enter 2nd coefficient: '))
    c = float(input('Enter 3rd coefficient: '))
    return a, b, c
def solve_and_output(a, b, c):
    disc = b*b - 4*a*c
    if disc >= 0:   # has real roots
        compute_real_roots(a, b, disc)
    else:           # has complex roots
        compute_complex_roots(a, b, disc)
```

Put new code here

```
def compute_real_roots(a, b, disc):
    if disc == 0:
        root = -b / (2*a)
        print(f'Only one real root: {root}')
    else:
        root1 = (-b + sqrt(disc)) / (2*a)
        root2 = (-b - sqrt(disc)) / (2*a)
        print(f'Two real roots: {root1} and {root2}')
```

```
def compute_complex_roots(a, b, disc):
    print("In compute_complex_roots:", a, b, disc) # dummy code
```

```
# ----- main ----- #
a, b, c = read_coefficients()
if a == 0:
    print("1st coefficient can't be zero. Program exits.")
    sys.exit()   # can't do anything more with bad input
solve_and_output(a, b, c)
```

Test Results

Test run

```
Enter 1st coefficient: 1
Enter 2nd coefficient: -4
Enter 3rd coefficient: 4
Only one real root: 2.0
```

discriminant is 0 (one real root)

Test run

```
Enter 1st coefficient: 2
Enter 2nd coefficient: 1
Enter 3rd coefficient: -1
Two real roots: 0.5 and -1.0
```

discriminant > 0 (two real roots)

Test run

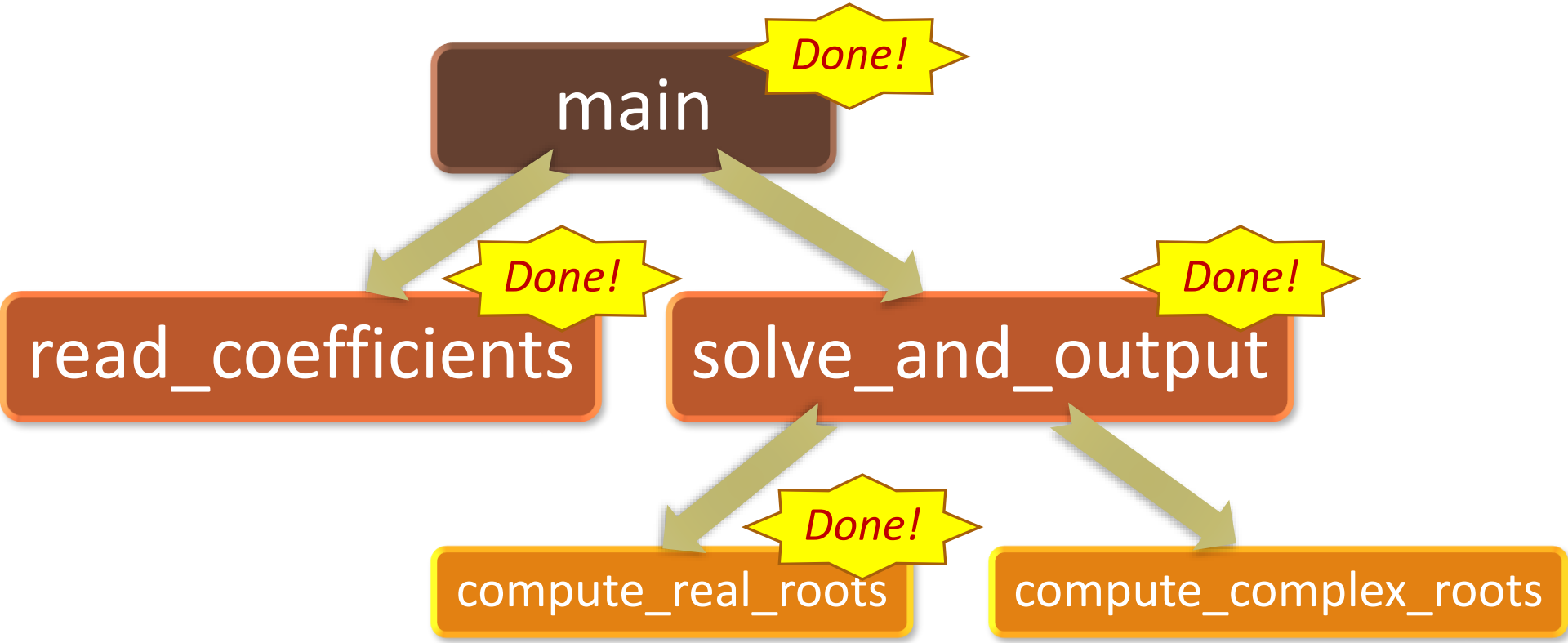
```
Enter 1st coefficient: 2
Enter 2nd coefficient: 1
Enter 3rd coefficient: 1
In compute_complex_roots: 2.0 1.0 -7.0
```

discriminant < 0 (complex root)

Our next task

So far so great !

What we've done so far



Next: Compute the complex roots

- The function compute_complex_roots() computes and prints the two complex roots.

```
def compute_complex_roots(a, b, disc):
```

$$\frac{-b}{2a} + i\frac{\sqrt{-D}}{2a}$$

D < 0

$$\frac{-b}{2a} - i\frac{\sqrt{-D}}{2a}$$

Now it's time for all good students to write it yourself!

Conclusion

- **Control structures** allow you to control the flow of your program's execution
- There are four fundamental control structures: *Sequence*, *Subroutine*, *Selection*, and *Repetition*. The previous chapters have already used the first two.
- The control structure *Selection* is used to select one of many possible paths of execution in a program depending on the given conditions. Each condition is expressed in Python by a *bool* expression.
- In Python, *Selection* can be expressed by the *if* statements or *if-else* statements. The *if* statement decides whether or not a code block is to be executed. The *if-else* statement selects between two possible code blocks to be executed.

References

- Comparison operations in Python:
  - https://docs.python.org/3/reference/expressions.html#comparisonsPython operators
- Boolean operations in Python:
  - https://docs.python.org/3/reference/expressions.html#boolean-operations
- Good tutorials for *if* and *if-else* statements:
  - http://interactivepython.org/runestone/static/thinkcspy/Selection/toctree.html
- Floating-point inexactness and rounding errors:
  - A digression on using floating points

Syntax Summary I

if statement

```
if condition:
    Code_Block
```

Condition
must be a ***bool*** expression.

A Code Block

```
    statement1
    statement2
    ...
    statementk
```

if-else statement

```
if condition:
    Code_Block1
else:
    Code_Block2
```

A code block
consists of one or more statements indented equally from the left.

Syntax Summary II : Python Operator Precedence

- From the highest precedence to the lowest down the table.
- Operators on the same row have the same precedence.

Category	Operators	Associativity
Subscription, call, attribute	`a[x]  f(x)  x.attribute`	left to right
Exponentiation	`**`	**right to left**
Unary sign	`+x  -x`	left to right
Multiplicative	`*  /  //  %`	left to right
Additive	`+  -`	left to right
Relational (comparison)	`==  !=  <  >  <=  >=`	left to right
Boolean NOT	`not`	left to right
Boolean AND	`and`	left to right
Boolean OR	`or`	left to right

Revision History

- August 2016 – Chalermsak Chatdokmaiprai
  - originally created for C#
- July 2017 – Chalermsak Chatdokmaiprai
  - adapted and enhanced for Python
- July 31, 2018 – Chalermsak Chatdokmaiprai
  - added examples of mysterious bugs caused by rounding errors
- January 5, 2020 – Chalermsak Chatdokmaiprai
  - expanded the topic of floating-point rounding errors
- January 12, 2020 – Chalermsak Chatdokmaiprai
  - minor improvement on the quadratic equation example

Constructive comments or error reports on this set of slides would be welcome and highly appreciated. Please contact Chalermsak.c@ku.ac.th