北京航空航天大學
BEIHANG UNIVERSITY

Lecture 2 :

Python Syntax Semantics Printing Decision

Xiaoqian Sun

Beihang University

Outline

- Algorithms and Programming languages
- Python
  - Variables
  - Input/Output
  - Named Constants
  - Operators
  - Making Decisions (if, elif, …)

Algorithms and Programming Languages

Algorithms

- General concept for describing the solution of a problem
  - A computational procedure …
- Many properties
  - Termination
  - Correctness / completeness
  - Time complexity
- Computer Science is about the formal study of algorithms
- Programs are implementations of algorithms
  - Algorithm: **Abstract**
  - Program: **Specific/Concrete**

Creating A Computer Program

‘Typical’ programmer

Translation

- A special computer program (translator) translates the program written by the programmer into the *only* form that the computer can understand (**machine language/binary**)

Program Creation

- A person (programmer) writes a computer program (series of instructions).
- The program is written and saved using a text editor/IDE.
- The instructions in the programming language (**e.g., Python**) are high level (look much like a human language).

```
a=2
print(a*3.1415926)
```

Execution

- The machine/binary language instructions can now be directly executed by the computer.

```
10000001
10010100 10000100
10000001 01010100
```

Types of Translators

1) **Interpreters** (e.g., Python is an interpreted language)
   - Each time the program is run the interpreter translates the program (translating a part at a time).
   - If there are any translation errors during the process of interpreting the program, the program will stop execution right when the error is encountered.

2) **Compilers** (e.g., 'C', C++ are compiled languages)
   - Before the program is run the compiler translates the program all at once.
   - If there are *any translation errors* during the compilation process, no machine language executable will be produced (nothing to execute)
   - If there are *no translation errors* during compilation then a machine language program is created which can then be executed.

Python - Overview

Python History

- Developed in the early 1990s by Guido van Rossum.
- Python was designed with a tradeoff in mind (from "*Python for everyone*" (Horstman and Necaise):
  - Pro: Python programmers could quickly *write programs* (and not be burdened with an overly difficult language)
  - Con: Python programs weren't optimized to *run* as efficiently as programs written in some other languages.
- Some advantages (from Python dot org)
  - Free
  - Powerful
  - Widely used (Google, NASA, Yahoo, Electronic Arts, some Linux operating system scripts etc.)
- Named after a British comedy "Monty Python's Flying Circus"
  - Official website (Python the programming language, not the Monty Python comedy troop): http://www.python.org

From: http://www.python.org/~guido/

"Gawky and proud of it."

The First Python Program

program1.py

```
a=2
b=3
print(a*b)
```

Running Programs

program1.py

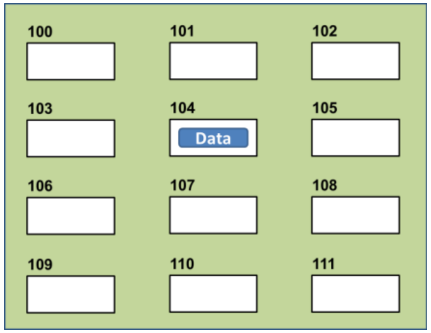
```
a=2
b=3
print(a*b)
```

- In general, three different ways:
  - Run whole script
    - Execute: python program1.py
    - Preset file written in text editor
  - Interactive environment
    - Execute: python
    - Type script line by line and inspect variables
    - Program is essentially developed over time
    - Convenient for simple debugging, testing, exploring data
  - Integrated development environment (IDE)
    - Open IDE, load file, run
    - Examples: Spyder, Ninja IDE, and many more
    - Very convenient for graphical debugging and testing

- Let’s have a quick look at these …
- We will do more of this in the lab session tomorrow!

Python - Syntax - Variables

Variables



- Set aside a location in memory.
- Used to store information (temporary).
- Some types of information which can be stored in variables include: integer (whole), floating point (fractional), strings (essentially any characters you can type and more)

Format (creating):

```
<name of variable> = <Information to be stored in the variable>
```

Examples (creating):

– Integer (e.g., `num1 = 10`)

– Floating point (e.g., `num2 = 10.0`)

– Strings: alpha, numeric, other characters enclosed in quotes.

  - e.g., `name = "james"`
  - To be safe get in the habit of using double (and not single) quotes

(Simple) data types in python

We start with three very simple datatypes for now:

- Integers:
  - 28, 5, 3, 2, 63464, 38957573484983 7538975, …
- Floats:
  - 0.352453, 32985.2349823, -9.0, …
- Strings:
  - “python”, “p”, …

The Assignment Operator: =

- The assignment operator '=' used in writing computer programs does not have the same meaning as mathematics.
  - Don’t mix them up!
- Example:

```
y = 3
x = y
x = 6
y = 13
```

- What is the end result?

Why does it make sense to have datatypes?

Why does it make sense to have datatypes?

- Datatypes provide often-used operators for working with variables of such a type
- Examples:
  - Addition, Multiplication
  - Concatenation
  - Removal
  - ….

Variable Naming Conventions

1. The name should be meaningful.

2. Names *must* start with a letter (Python requirement) and *should not* begin with an underscore (style requirement).

3. Names are case sensitive but avoid distinguishing variable names only by case.
4. Can't be a keyword (see next slide).

Key Words In Python[1]

and
as
assert
break
class
continue
def
del
elif
else
except
exec
finally
for
from
global
if
import
in
is
lambda
not
or
pass
print
raise
return
try
while
with
yield

[1] From “*Starting out with Python*” by Tony Gaddis

Python - Syntax – Input/Output

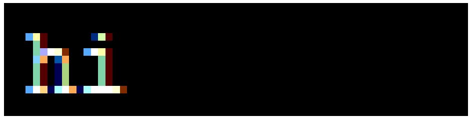
What is syntax?

- **Syntax** refers to how some thing is denoted
  - It is about the form/shape of elements
- This is different fromt he meaning, which is **Semantic**
  - It is about the associatio/meaning in somebody‘s head
- Example:
  - Syntax: *„What time is it?"* (English)
  - Semantics: *Somebody wants to kow the current time.* (Meaning)
- Natural languages often have different syntax, but highly similar semantics (concepts)
  - Example?

Displaying Output Using The `print()` Function

- This function takes zero or more arguments (inputs)
  - Multiple arguments are separated with commas
  - `print()` will display all the arguments followed by a blank line (move the cursor down a line).
- Simple Example:

```
print("hi")
```

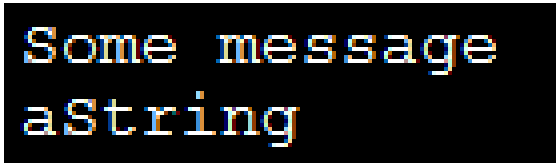


```
hi
```

print("… ") Vs. print(<*name*>)

- Enclosing the value in brackets with quotes means the value in between the quotes will be literally displayed onscreen.
- Excluding the quotes will display the contents of a memory location.
- Example:

```
aString = "Some message"
print(aString)
print("aString")
```



```
Some message
aString
```

Printing multiple things

- Simply separate things by a comma
- Examples:

```
name = “John"
print(“My name is”,name)
print(name,name)
print(“The value of x is”,x)
```

Input

- The computer program getting *string information* from the user.
- Strings cannot be used for calculations (information for getting numeric input will provided shortly).

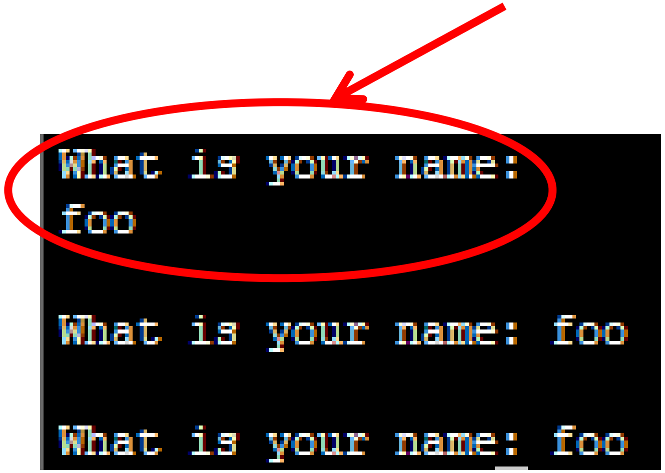
- **Format:**

```
<variable name> = input()
          OR
<variable name> = input("<Prompting message>")
```

- **Example:**

```
print("What is your name: ")
name = input()
         OR
name = input("What is your name: ")
         OR
print("What is your name: ", end="")
name = input()
```

Avoid alignment issues such as this



```
What is your name:
foo

What is your name: foo

What is your name: foo
```

Python – Named Constants

Named Constants

- They are similar to variables:
  - A memory location that's been given a name.
- Unlike variables their contents *shouldn't* change.
- The naming conventions for choosing variable names generally apply to constants but the name of constants should be all UPPER CASE. (You can separate multiple words with an underscore).
- Example **PI** = 3.14
  - PI = Named constant, 3.14 = Unnamed constant
- They are capitalized so the reader of the program can distinguish them from variables.
  - For some programming languages the translator will enforce the unchanging nature of the constant.
  - For languages such as *Python it is up to the programmer* to recognize a named constant and not to change it.

Why Use Named Constants

1. They make your program easier to read and understand

```
# NO
populationChange = (0.1758 – 0.1257) * currentPopulation
```

Avoid unnamed constants whenever possible!

Vs.

```
#YES
BIRTH_RATE = 17.58
MORTALITY_RATE = 0.1257
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) *
   currentPopulation
```

Why Use Named Constants (2)

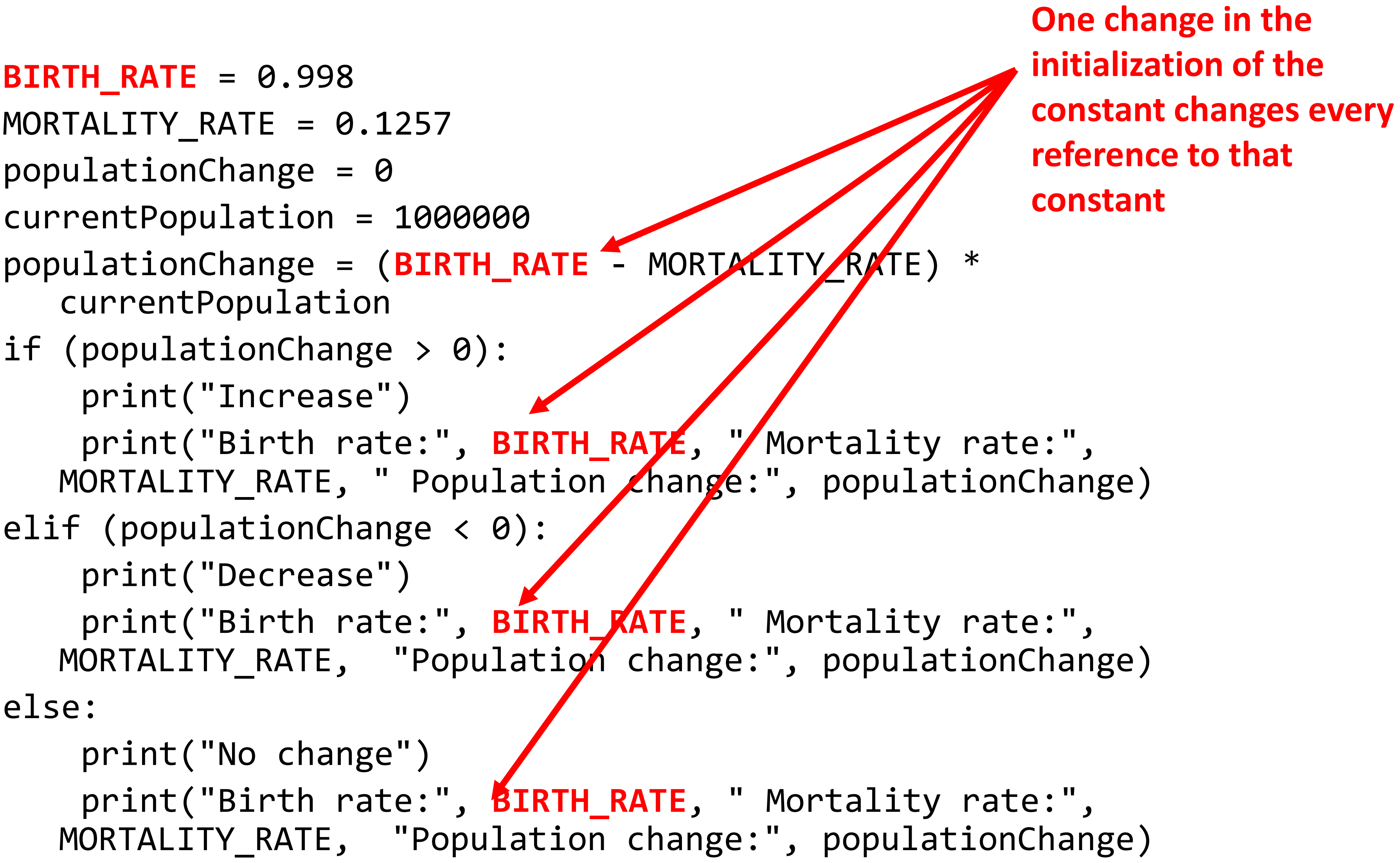
2) Makes the program easier to maintain

- If the constant is referred to several times throughout the program, changing the value of the constant once will change it throughout the program.
- Using named constants is regarded as “good style” when writing a computer program.

Purpose Of Named Constants (3)

```
BIRTH_RATE = 0.998
MORTALITY_RATE = 0.1257
populationChange = 0
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) *
   currentPopulation
if (populationChange > 0):
    print("Increase")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
   MORTALITY_RATE, " Population change:", populationChange)
elif (populationChange < 0):
    print("Decrease")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
   MORTALITY_RATE,  "Population change:", populationChange)
else:
    print("No change")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
   MORTALITY_RATE,  "Population change:", populationChange)
```

Purpose Of Named Constants (4)



```
BIRTH_RATE = 0.998
MORTALITY_RATE = 0.1257
populationChange = 0
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) *
   currentPopulation
if (populationChange > 0):
    print("Increase")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
   MORTALITY_RATE, " Population change:", populationChange)
elif (populationChange < 0):
    print("Decrease")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
   MORTALITY_RATE,  "Population change:", populationChange)
else:
    print("No change")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
   MORTALITY_RATE,  "Population change:", populationChange)
```

One change in the initialization of the constant changes every reference to that constant

Purpose Of Named Constants (5)

```
BIRTH_RATE = 0.1758
MORTALITY_RATE = 0.0001
populationChange = 0
currentPopulation = 1000000
populationChange = (BIRTH_RATE - MORTALITY_RATE) *
   currentPopulation
if (populationChange > 0):
    print("Increase")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
   MORTALITY_RATE, " Population change:", populationChange)
elif (populationChange < 0):
    print("Decrease")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
   MORTALITY_RATE,  "Population change:", populationChange)
else:
    print("No change")
    print("Birth rate:", BIRTH_RATE, " Mortality rate:",
   MORTALITY_RATE,  "Population change:", populationChange)
```

One change in the initialization of the constant changes every reference to that constant

Python – Operators and Conversion

This is about working with variables!

An operator modifies a variable (or more than one) => operation on variables

Arithmetic Operators

Operator	Description	Example
=	Assignment	num = 7
+	Addition	num = 2 + 2
-	Subtraction	num = 6 - 4
*	Multiplication	num = 5 * 4
/	Division	num = 9 / 2 4.5
//	Integer division	num = 9 // 2 4
%	Modulo	num = 9 % 2 1
**	Exponent	num = 9 ** 2 81

Order Of Operation

- First level of precedence: top to bottom
- Second level of precedence
  - If there are multiple operations that are on the same level then precedence goes from left to right.

`()`	Brackets (inner before outer)
`**`	Exponent
`*, /, //, %`	Multiplication, division, modulo
`+, -`	Addition, subtraction
`=`	Assignment

Example

```
x = 5 * 2 ** 3

Vs.
x = (5 * 2) ** 3
```

Order Of Operation And Style

- Even for languages where there are clear rules of precedence (e.g., Java, Python) it’s good style to explicitly bracket your operations and use blank spaces as separators.

```
x = (a * b) + (c / d)
```

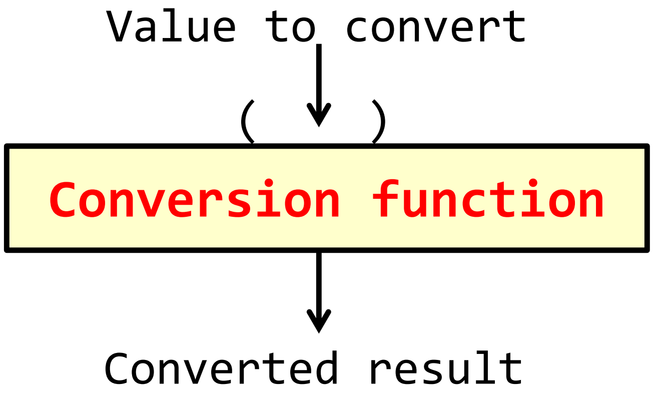
- It not only makes it easier to read complex formulas but also a good habit for languages where precedence is not always clear (e.g., C++, C).

Converting Between Different Types Of Information

- Example motivation: you may want numerical information to be stored as a string (for built in string functions e.g., check if a string consists only of numbers) but also you want to perform calculations).
- Some of the conversion mechanisms (functions) available in Python:

Format:

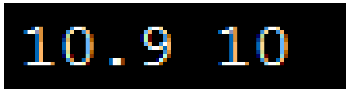
```
int(<value to convert>)
float(<value to convert>)
str(<value to convert>)
```



Examples:

Program name: `convert1.py`

```
x = 10.9
y = int(x)
print(x,y)
```



```
10.9 10
```

Converting Types: Extra Practice

- Determine the output of the following program:

```
print(12+33)
print('12'+'33')
x = 12
y = 21
print(x+y)
print(str(x)+str(y))
```

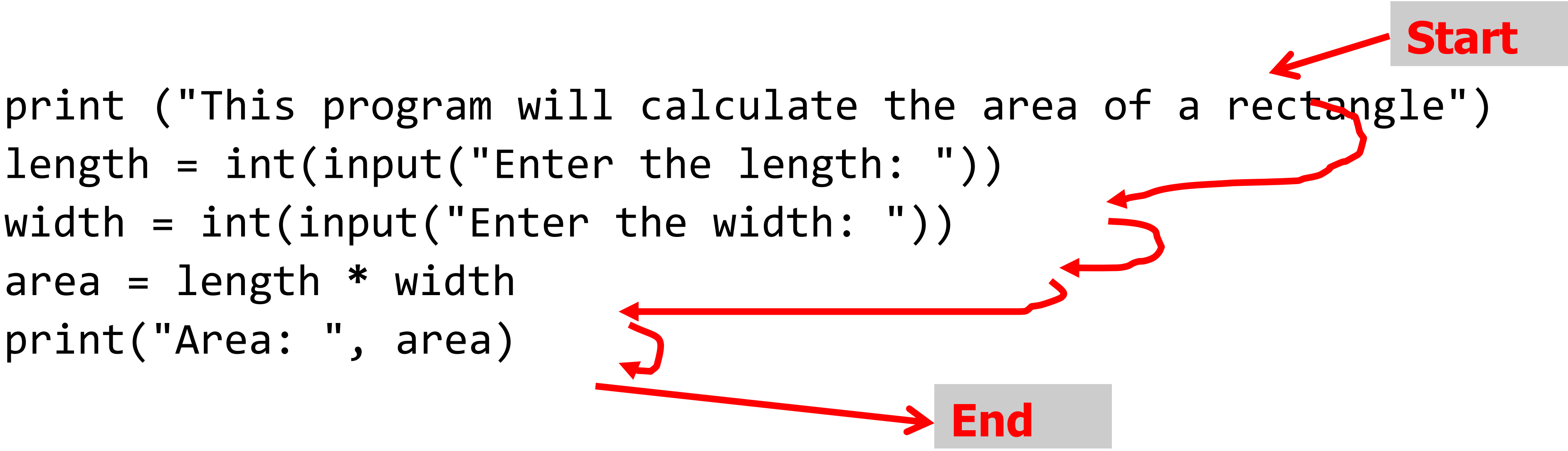
Section Summary: Input, Representations

- How to get user input in Python
- How do the different types of variables store/represent information (optional/extra for now)
- How/why to convert between different types

Making Decisions In Python

In this section you will learn how to have your programs choose between alternative courses of action.

Recap: Programs You’ve Seen So Far Produces Sequential Execution



Start

```
print ("This program will calculate the area of a rectangle")
length = int(input("Enter the length: "))
width = int(input("Enter the width: "))
area = length * width
print("Area: ", area)
```

End

Programming: Decision Making Is Branching

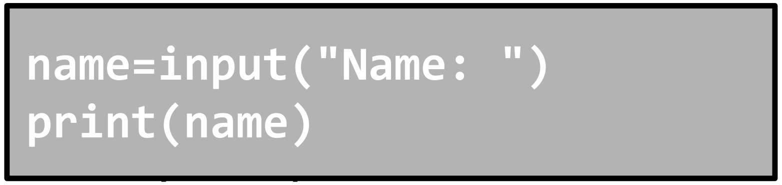
- Decision making is choosing among alternates (branches).
- Why is it needed?
  - When alternative courses of action are possible and each action may produce a different result.
- In terms of a computer program the choices are stated in the form of a question that only yield an answer that is either true or false
  - Although the approach is very simple, modeling decisions in this fashion is a very useful and powerful tool.

Decision-Making In Programming (Python)

- Decisions are questions with answers that are either true or false (Boolean expressions) e.g., Is it true that the variable ‵num′ is positive?
- The program may branch one way or another depending upon the answer to the question (the result of the Boolean expression).
- Decision making/branching constructs (mechanisms) in Python:
  - `If` (reacts differently only for true case)
  - `If-else` (reacts differently for the true or false cases)
  - `If-elif-else` (multiple cases possible but only one case can apply, if one case is true then it's false that the other cases apply)

New Terminology

- **Boolean expression**: An expression that must work out (evaluate to) to either a true or false value.
  - e.g., it is over 45 Celsius today
  - e.g., the user correctly entered the password
- **New term, body**: A block of program instructions that will execute under a specified condition (for branches the body executes when the Boolean expression evaluates to/works out to true)

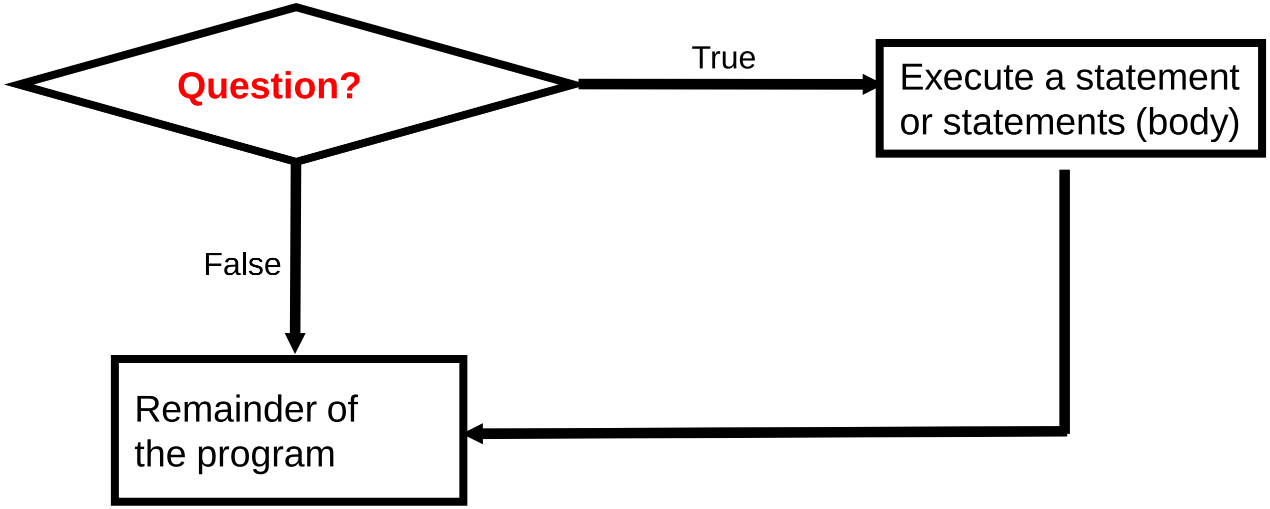


```
name=input("Name:  ")
print(name)
```

- The 'body' is indented (4 spaces)

This/these instruction/instructions run when you give the Python interpreter the name of a file, the 'body' of the Python program runs

Decision Making With An 'If'



The ‘If’ Construct

- Decision making: checking if a condition is true (in which case something should be done).
- **Format:**

```
(General format)
   if (Boolean expression):
       body
(Detailed structure)
if (<operand> <relational operator> <operand>):
     body
```

Boolean expression

Note: Indenting the body is mandatory!

The ‘If’ Construct (2)

- **Example:**

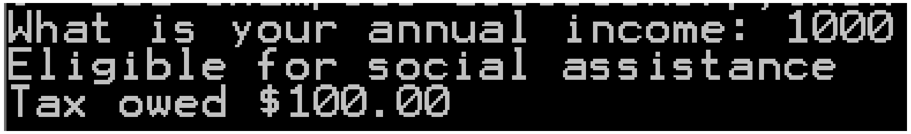
```
age = int(input("Age: "))
if (age >= 18):
   print("You are an adult")
```

Note On Indenting (1)

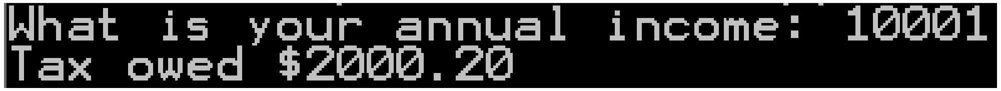
- In Python indenting is mandatory in order to determine which statements are part of a body (**syntactically required** in Python).

```
# Single statement body
if (num == 1):
    print("Body of the if")
print("After body")

# Multi-statement body (program ‘if2.py’)
taxCredit = 0
taxRate = 0.2
income = float(input("What is your annual income: "))
if (income < 10000):
     print("Eligible for social assistance")
     taxCredit = 100
tax = (income * taxRate) – taxCredit
print("Tax owed $“,tax)
```



```
What is your annual income: 1000
Eligible for social assistance
Tax owed $100.00
```



```
What is your annual income: 10001
Tax owed $2000.20
```

Note On Indenting (2)

- A “sub-body” (`IF`-branch) is indented by an additional 4 spaces (8 or more spaces) if one `IF`-branch is inside the body of another `IF`-branch (this is called ‘nesting’ – more details later).
  - Usually, you simply press TAB key one time for indentation

New Terminology

- **Operator/Operation**: action being performed
- **Operand**: the item or items on which the operation is being performed.

Examples:

```
2 + 3
2 * (-3)
```

Allowable Operands For Boolean Expressions

Format:

```
    if (operand     relational operator    operand):
```

Example:

```
    if (age >= 18):
```

Some operand types

- integer
- floats
- String
- Boolean (True or False)

Make sure that you are comparing operands of the same type or at the very least they must be comparable!

Allowable Relational Operators For Boolean Expressions

```
if (operand    relational operator    operand) then
```

Python operator	Mathematical equivalent	Meaning	Example
<	<	Less than	5 < 3
>	>	Greater than	5 > 3
==	=	Equal to	5 == 3
<=	≤	Less than or equal to	5 <= 5
>=	≥	Greater than or equal to	5 >= 4
!=	≠	Not equal to	x != 5

Common Mistake

- Do not confuse the equality operator '==' with the assignment operator '='.
- **Example (Python syntax error)**[1]:

```
if (num = 1):    # Not the same as   if (num == 1):
```

  To be extra safe some programmers put unnamed constants on the left hand side of an equality operator (which always/almost always results in a syntax error rather than a logic error if the assignment operator is used in place of the equality operator).

- A way of producing syntax rather than a logic error:

```
if (1 = num)
```

[1] This not a syntax error in all programming languages so don't get complacent and assume that the language will automatically "take care of things" for you.

An Application Of Branches

- Branching statements can be used to check the validity of data (if the data is correct or if the data is a value that's allowed by the program).
- **General structure**:

```
if (error condition has occurred)
    React to the error (at least display an error message)
```

- **Example**:

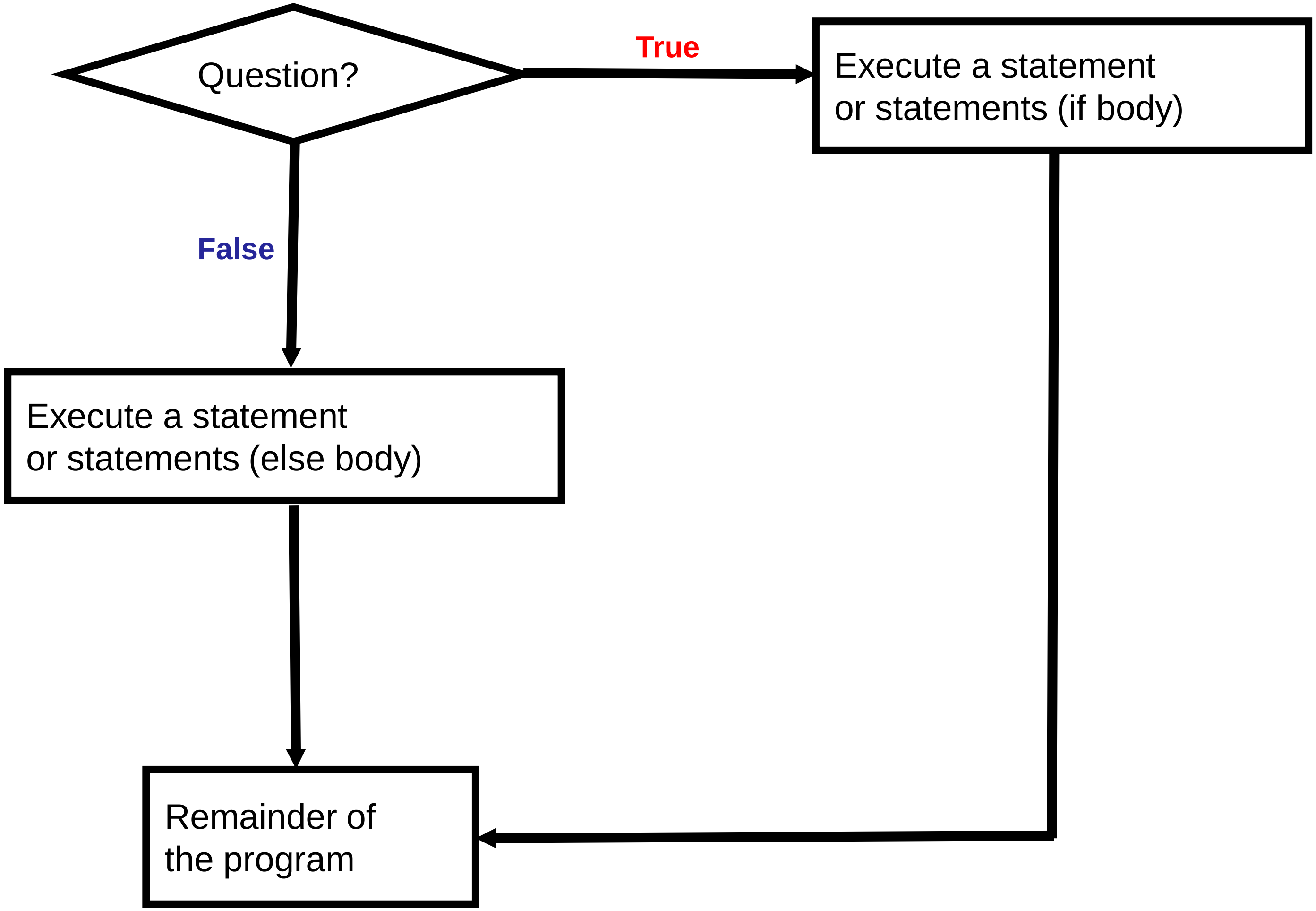
```
if (age < 0):
    print("Age cannot be a negative value")
```

Tip: if data can only take on a certain value (or range) do not automatically assume that it will be valid. Check the validity of range before proceeding onto the rest of the program.

Decision Making With An 'If': Summary

- Used when a question (Boolean expression) evaluates only to a true or false value (Boolean):
  - If the question evaluates to true then the program reacts differently. It will execute the body after which it proceeds to the remainder of the program (which follows the if construct).
  - If the question evaluates to false then the program doesn't react differently. It just executes the remainder of the program (which follows the if construct).

Decision Making With An ‘If-Else’


Question?
True
False
Execute a statement or statements (if body)
Execute a statement or statements (else body)
Remainder of the program

The `If-Else` Construct

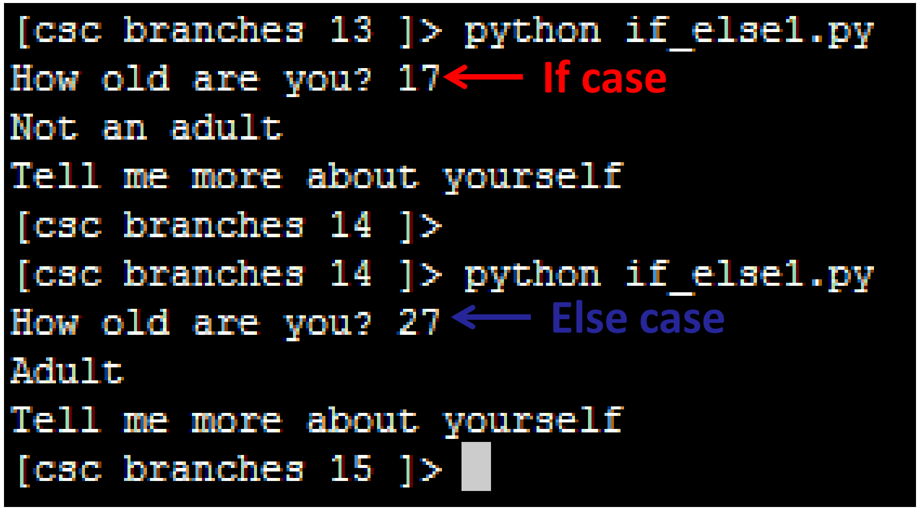
- Decision making: checking if a condition is true (in which case something should be done) but unlike '`if`' *also reacting if the condition is not true (false).*
- **Format:**

```
if (operand  relational operator  operand):
    body of 'if'
else:
    body of 'else'
additional statements
```

If-Else Construct (2)

- **Partial example:**

```
if (age < 18):
    print("Not an adult")
else:
    print("Adult")
print("Tell me more about yourself")
```



```
[csc branches 13 ]> python if_else1.py
How old are you? 17
Not an adult
Tell me more about yourself
[csc branches 14 ]>
[csc branches 14 ]> python if_else1.py
How old are you? 27
Adult
Tell me more about yourself
[csc branches 15 ]>
```

Quick Summary: `If` Vs. `If-Else`

- If:
  - Evaluate a Boolean expression (ask a question).
  - If the expression evaluates to true then execute the 'body' of the `if`.
  - No additional action is taken when the expression evaluates to false.
  - Use when your program is supposed to react differently only when the answer to a question is true (and do nothing different if it's false).
- If-Else:
  - Evaluate a Boolean expression (ask a question).
  - If the expression evaluates to true then execute the 'body' of the `if`.
  - If the expression evaluates to false then execute the 'body' of the `else`.
  - That is: *Use when your program is supposed to react differently for both the true and the false cases*.

Logical Operations

- There are many logical operations but the three most commonly used in computer programs include:
  - Logical AND
  - Logical OR
  - Logical NOT

Logical AND

- The popular usage of the logical AND applies when *ALL* conditions must be met.
- Example:
  - Pick up your son AND pick up your daughter after school today.

Condition I

Condition II

- Logical AND can be specified more formally in the form of a truth table.

Truth table (AND)

C1	**C2**	**C1 AND C2**
False	False	False
False	True	False
True	False	False
True	*True*	*True*

Logical AND: Three Input Truth Table

Truth table

C1	C2	C3	C1 AND C2 AND C3
False	False	False	False
False	False	True	False
False	True	False	False
False	True	True	False
True	False	False	False
True	False	True	False
True	True	False	False
True	*True*	*True*	*True*

Evaluating Logical AND Expressions

- True **AND** True **AND** True
- False **AND** True **AND** True
- True **AND** True **AND** True **AND** False

Logical OR

- The correct everyday usage of the logical OR applies when *ATLEAST* one condition must be met.
- Example:
  - You are using additional recommended resources for this course: the online textbook OR the paper textbook available in the bookstore.

Condition I

Condition II

- Similar to AND, logical OR can be specified more formally in the form of a truth table.

Truth table		
C1	**C2**	**C1 OR C2**
False	*False*	*False*
False	True	True
True	False	True
True	True	True

Logical OR: Three Input Truth Table

Truth table

C1	**C2**	**C3**	**C1 OR C2 OR C3**
False	*False*	*False*	*False*
False	False	True	True
False	True	False	True
False	True	True	True
True	False	False	True
True	False	True	True
True	True	False	True
True	True	True	True

Evaluating Logical OR Expressions

- True **OR** True **OR** True
- False **OR** True **OR** True
- False **OR** False **OR** False **OR** True

Logical NOT

- The everyday usage of logical NOT negates (or reverses) a statement.
- Example:
  - I am finding this class quite stimulating and exciting .....*NOT!!!*

Statement (logical condition)

Negation of the statement/condition

- The truth table for logical NOT is quite simple:

Truth table	
S	**Not S**
False	True
True	False

Evaluating More Complex Logical Expressions

- Order of operation (left to right evaluation if the ‘level’ is equal)
  1. Brackets (inner first)
  2. Negation
  3. AND
  4. OR

Evaluating More Complex Logical Expressions

- True **OR** True **AND** True
- **NOT** (False **OR** True) **OR** True
- (False **AND** False) **OR** (False **AND** True)
- **NOT NOT NOT NOT** True
- **NOT NOT NOT** False

Extra Practice

Assume the variables a = 2, b = 4, c = 6

For each of the following conditions indicate whether the final value is true or false.

Expression	Final result
a == 4 or b > 2	
6 <= c and a > 3	
1 != b and c != 3	
a >-1 or a <= b	
not (a > 2)	

Logic Can Be Used In Conjunction With Branching

- Typically the logical operators AND, OR are used with multiple conditions/Boolean expressions:
  - If multiple conditions *must all be met* before the body will execute. (AND)
  - If *at least one condition* must be met before the body will execute. (OR)
- The logical NOT operator can be used to check for inequality (not equal to).
  - E.g., If it's true that the user *did not* enter an invalid value the program can proceed.

The “NOT” Operator

- **Format:**

```
if not (Boolean expression):
    body
```

- **Name of the online example:** `if_not.py`
  - (An equivalent solution can be implemented using the inequality operator ‘!=‘)

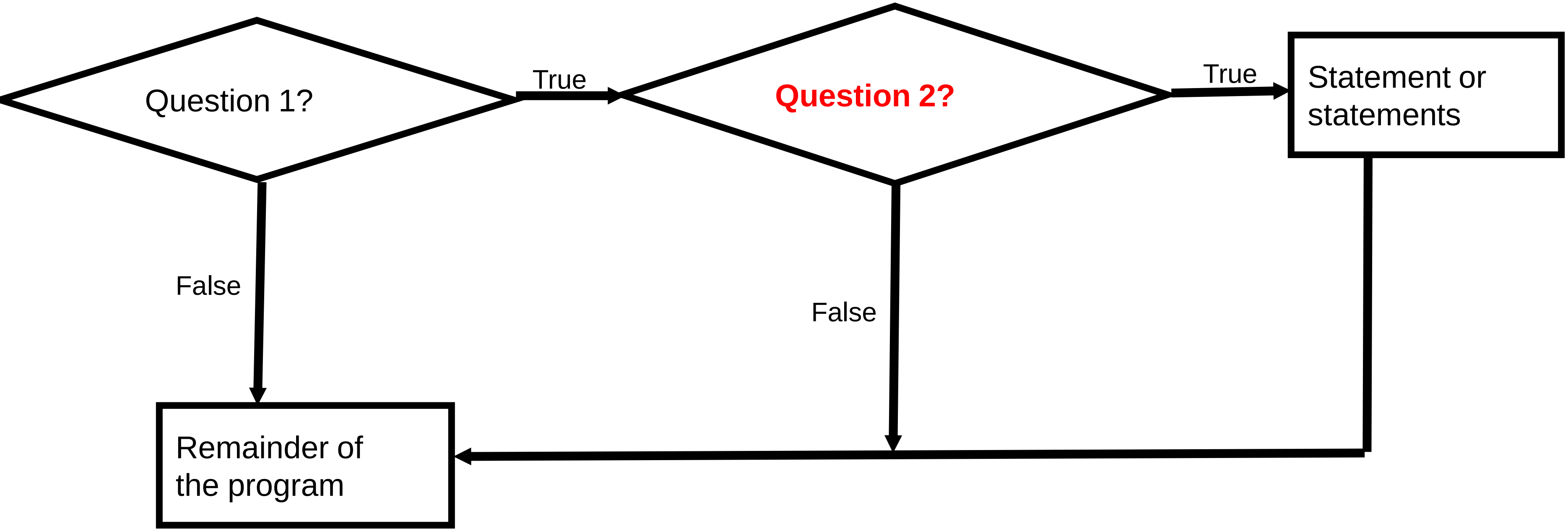
```
SYSTEM_PASSWORD = "password123"
userPassword = input("Password: ")
if not (userPassword == SYSTEM_PASSWORD):
    print("Using logical NOT-operator: Wrong password")
```

Quick Summary: Using Multiple Expressions

- Use multiple expressions when multiple questions must be asked and the result of expressions are related:
- AND (strict: all must apply):
  - All Boolean expressions must evaluate to true before the entire expression is true.
  - If any expression is false then whole expression evaluates to false.
- OR (less restrictive: at least one must apply):
  - If any Boolean expression evaluates to true then the entire expression evaluates to true.
  - All Boolean expressions must evaluate to false before the entire expression is false.
- Not:
  - Negates or reverses the logic of a Boolean expression
  - May sometimes be super ceded by the use of an inequality operator

Nested Decision Making

- Decision making is dependent.
- The first decision must evaluate to true (“gate keeper”) before successive decisions are even considered for evaluation.

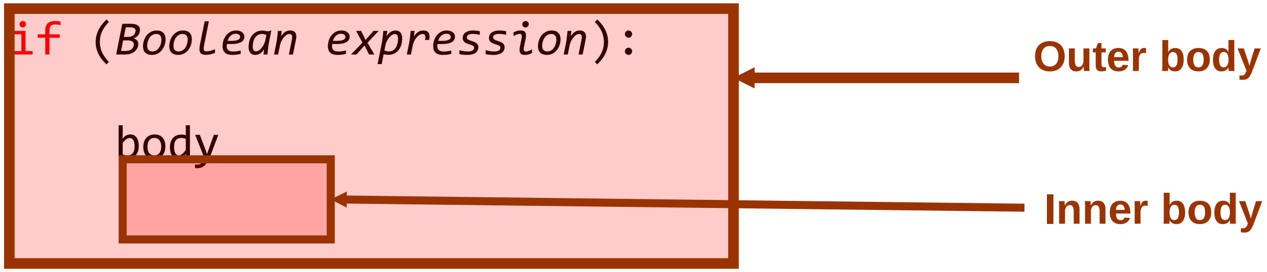


What might that look like in Python?

Nested Decision Making

- One decision is made inside another.
- Outer decisions must evaluate to true before inner decisions are even considered for evaluation.
- **Format:**

```
   if (Boolean expression):
```



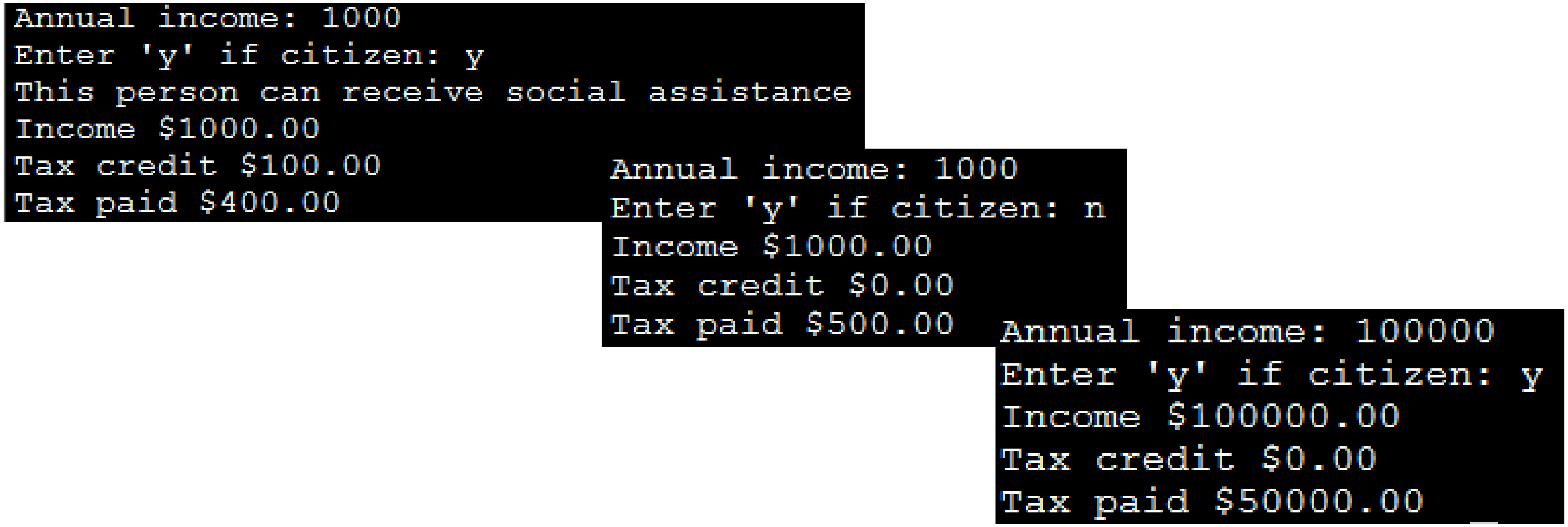
```
if (Boolean expression):

     body
```

Nested Decision Making (2)

- **Partial example:** `nesting.py`

```
if (income < 10000):
    if (citizen == 'y'):
        print("This person can receive social assistance")
        taxCredit = 100
tax = (income * TAX_RATE) - taxCredit
```



```
Annual income: 1000
Enter 'y' if citizen: y
This person can receive social assistance
Income $1000.00
Tax credit $100.00
Tax paid $400.00
```

```
Annual income: 1000
Enter 'y' if citizen: n
Income $1000.00
Tax credit $0.00
Tax paid $500.00
```

```
Annual income: 100000
Enter 'y' if citizen: y
Income $100000.00
Tax credit $0.00
Tax paid $50000.00
```

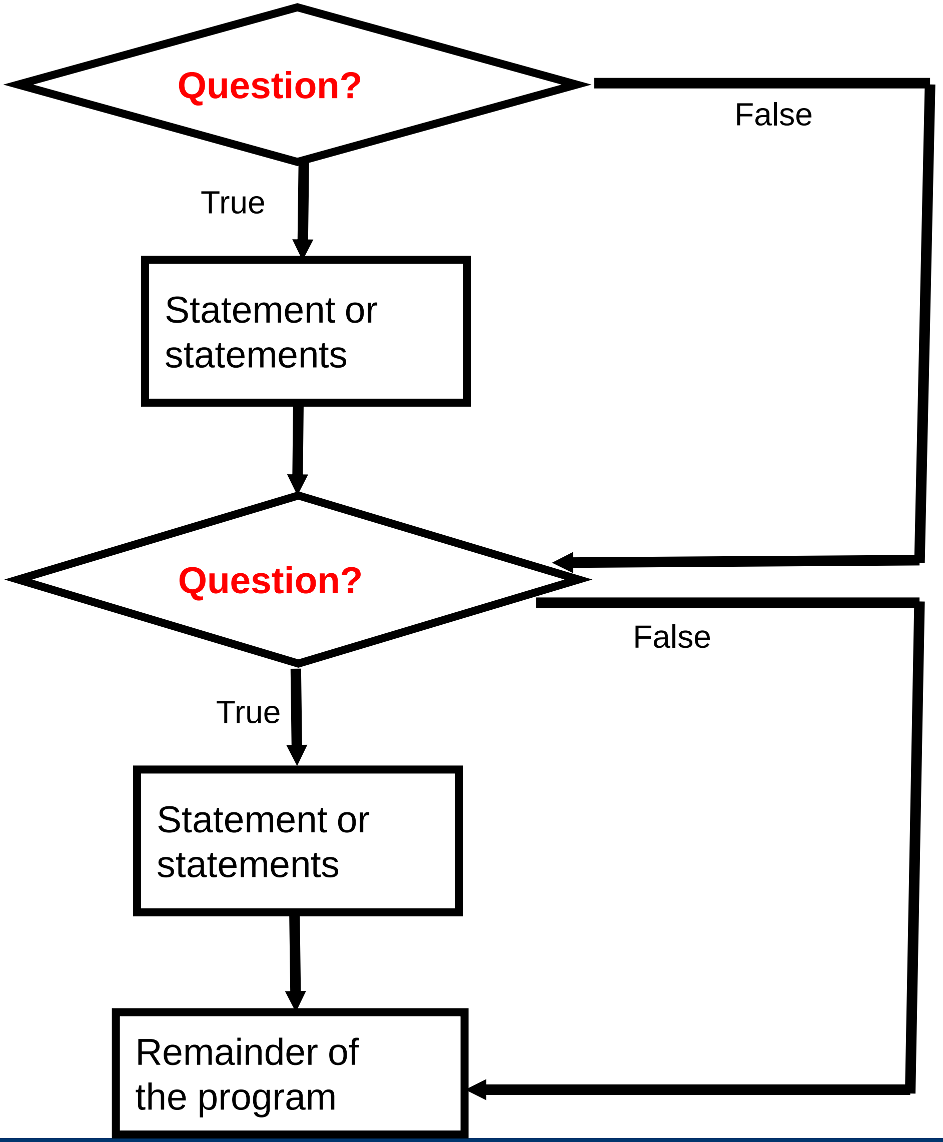
Question

- What’s the difference between employing nested decision making and a logical AND?

Decision-Making With Multiple Alternatives/Questions

- `IF` (single question)
  - Checks a condition and executes a body if the condition is true
- `IF-ELSE` (single question)
  - Checks a condition and executes one body of code if the condition is true and another body if the condition is false
- Approaches for multiple (two or more) questions
  - **Multiple IF's**
  - **IF-ELIF-ELSE**

Decision Making With Multiple If's



Multiple If's: Non-Exclusive Conditions

- Any, all or none of the conditions may be true (independent)
- Employ when a series of independent questions will be asked
- **Format:**

```
if (Boolean expression 1):
     body 1
if (Boolean expression 2):
     body 2
          :
statements after the conditions
```

Multiple If's: Mutually Exclusive Conditions

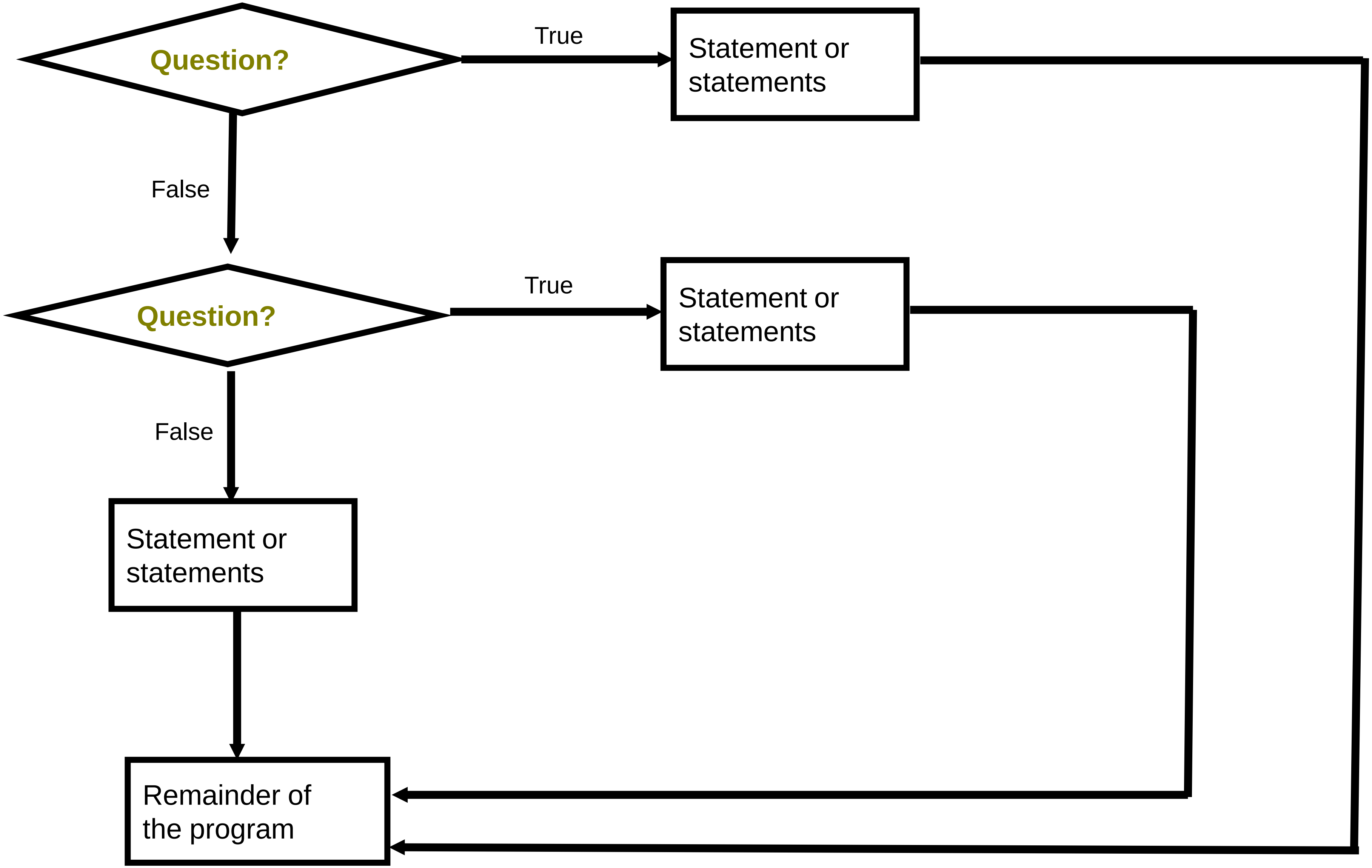
- At most *only one* of many conditions can be true
- Can be implemented through multiple if's

Inefficient combination!

- **Example**: The name of the complete online program is: "grades_inefficient.py"

```
if (gpa == 4):
    letter = 'A'
if (gpa == 3):
    letter = 'B'
if (gpa == 2):
    letter = 'C'
if (gpa == 1):
    letter = 'D'
if (gpa == 0):
    letter = 'F'
```

Decision Making With `If-Elif-Else`



Multiple If-Elif-Else: Use With Mutually Exclusive Conditions

- **Format:**

```
if (Boolean expression 1):
     body 1
elif (Boolean expression 2):
     body 2
          :
else
     body n
statements after the conditions
```

Mutually exclusive

•One condition evaluating to true excludes other conditions from being true

•Example: having your current location as ‘Calgary’ excludes the possibility of the current location as ‘Edmonton’, ‘Toronto’, ‘Medicine Hat’

If-Elif-Else: Mutually Exclusive Conditions (Example)

- **Example**: The name of the complete online program is: “grades_efficient.py”

```
if (gpa == 4):
    letter = 'A'

elif (gpa == 3):
    letter = 'B'

elif (gpa == 2):
    letter = 'C'

elif (gpa == 1):
    letter = 'D'

elif (gpa == 0):
    letter = 'F'
else:
    print("GPA must be one of '4', '3', '2', '1' or '1'“)
```

This approach is more efficient when at most only one condition can be true.

Extra benefit:

The body of the else executes only when all the Boolean expressions are false. (Useful for error checking/handling).

When To Use `Multiple-Ifs`

- When all conditions must be checked (more than one Boolean expressions for each ‘if’ can be true).
  - Non-exclusive conditions
- Example:
  - Some survey questions:
    - When all the questions must be asked
    - The answers to previous questions will not affect the asking of later questions
      - E.g.,
      - Q1: What is your gender?
      - Q2: What is your age?
      - Q3: What is your country of birth?

When To Use `If, ElIfs`

- When all conditions may be checked but at most only one Boolean expression can evaluate to true.
  - Exclusive conditions
- Example:
  - Survey questions:
    - When only some of the questions will be asked
    - The answers to previous questions WILL affect the asking of later questions
      - E.g.,
      - Q1: Were you born in BC?
      - Q2 (ask only if the person answered 'no' to the previous): Were you born in AB?
      - Q3 (ask only if the person answered 'no' to the previous questions): Were you born in SK?
      - …

Recap: What Decision Making Mechanisms Are Available /When To Use Them

Mechanism	When To Use
`If`	Evaluate a Boolean expression and execute some code (body) if it’s true
`If-else`	Evaluate a Boolean expression and execute some code (first body: ‘if’) if it’ s true, execute alternate code (second body: ‘else’) if it’s false
`Multiple if’ s`	Multiple Boolean expressions need to be evaluated with the answer for each expression being independent of the answers for the others (non-exclusive). Separate instructions (bodies) can be executed for each expression.
`If-elif-else`	Multiple Boolean expressions need to be evaluated but zero or at most only one of them can be true (mutually exclusive). Zero bodies or exactly one body will execute. Also it allows for a separate body (`else`-case) to execute when all the `if-elif` Boolean expressions are false.

Thank you very much!