SEE1002
Introduction to Computing for Energy and Environment

Part 2: Elements of Python programming

Sec. 3: Branching or Decision Making

Course Outline

Part 1: Introduction to computing

Part 2: Elements of Python programming

Section 1: Data and variables

Section 2: Elementary data structures

Section 3: Branching

Section 4: Loops

Section 5: Functions

Part 3: Basic Python programming

Section 1: Structure of a Python program

Section 2: Input and output

Section 3: Modules

Section 4: Good programming practices

Part 4: Python for science and engineering

Section 1: Vectors, matrices and arrays

Section 2: NumPy and SciPy

Objectives

1. Understand meaning of program flow
2. Make simple and complicated decisions in Python

Outline

1. Program flow
2. Branching or decision making
3. Extensions

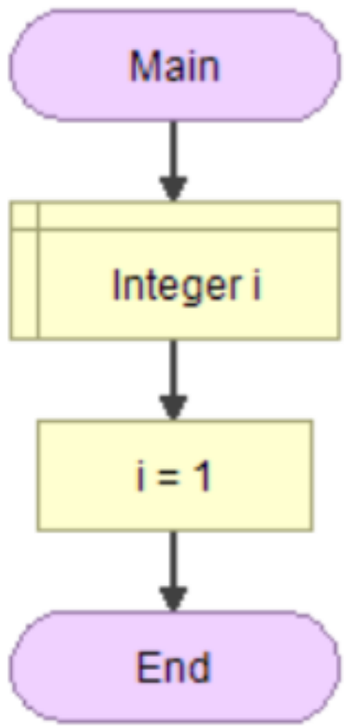
1. Program flow

Introduction

- We are now ready to start writing Python programs using the editor.
- Up till now, we've been focusing on the shell. In the shell, the statements are executed in the order that they are typed
- What happens with real programs? This leads to us the topic of program flow.

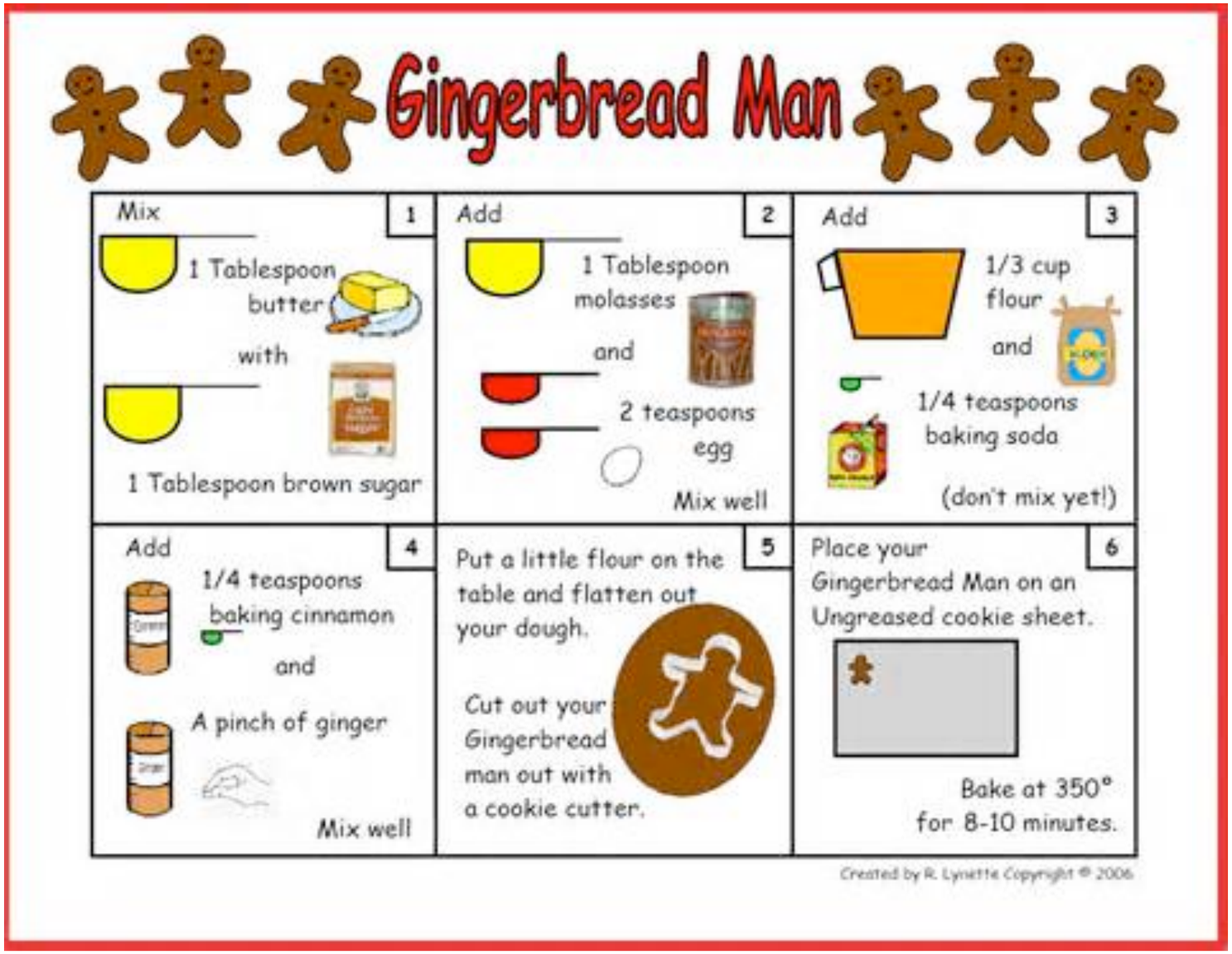
What is program flow?

- In its simplest form a computer program is nothing more than a sequence of instructions.
- Program flow refers to the *sequence of steps*, or alternatively, the way in which tasks or boxes are connected to each other.
- We will consider various type of program flow in this course.



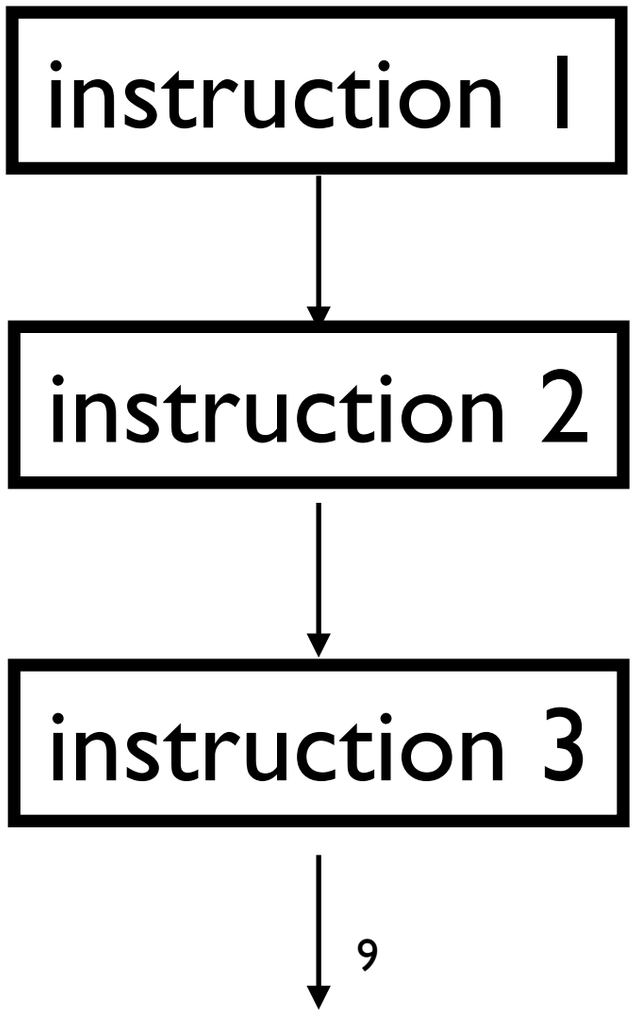
Concrete example

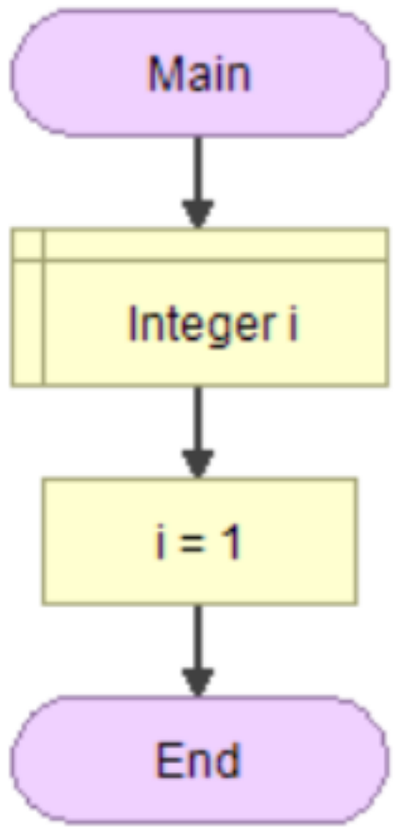
Previously we emphasised that a recipe is like an algorithm in a program.



Sequential flow

- The simple programs rely on linear or sequential flow.
- It can be represented graphically as:

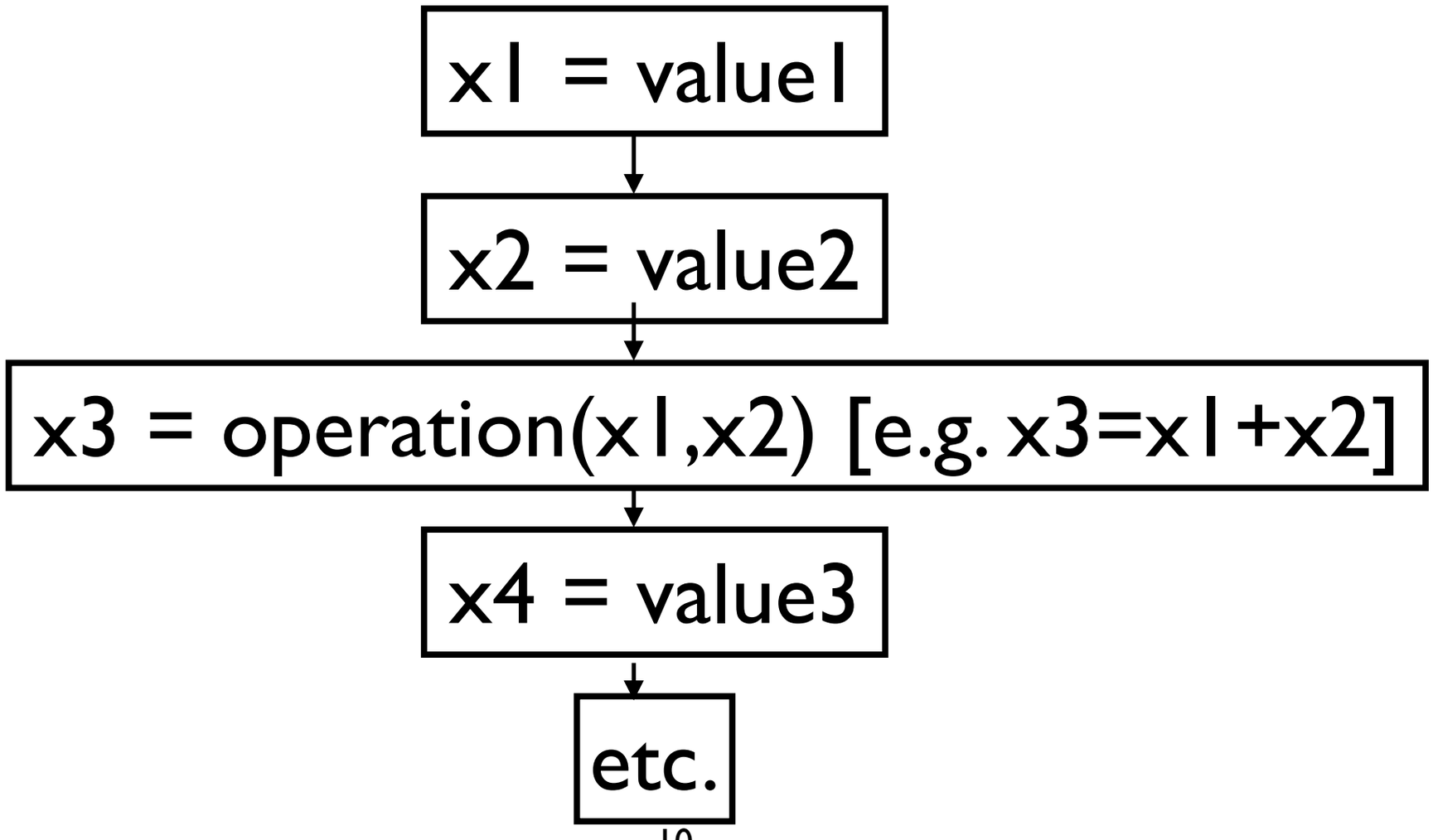




Implementation

Simple computer programs can be written with a completely linear program flow.

Example:



Limitations

- With sequential flow, the values can be changed manually but the operations always have to be carried out in the same order.
- This is a severe limitation. It excludes the vast majority of useful programs.

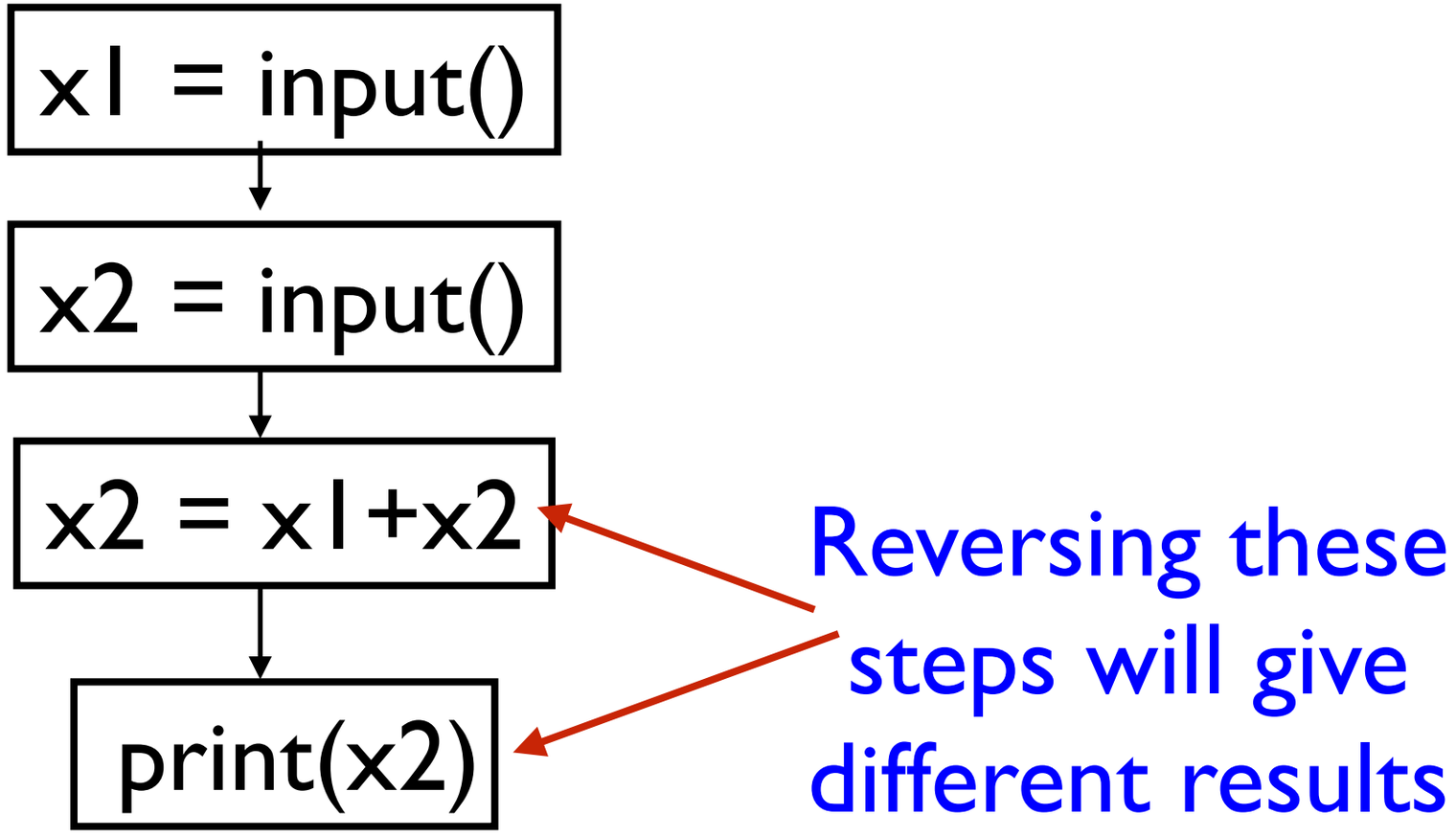


Illustration in Python

```
x1=int(input('Enter x1: '))
x2=int(input('Enter x2:' ))
x2=x1+x2
print(x2)
```

```
Enter x1: 2

Enter x2:3
5
```

```
x1=int(input('Enter x1: '))
x2=int(input('Enter x2:' ))

print(x2)
x2=x1+x2
```

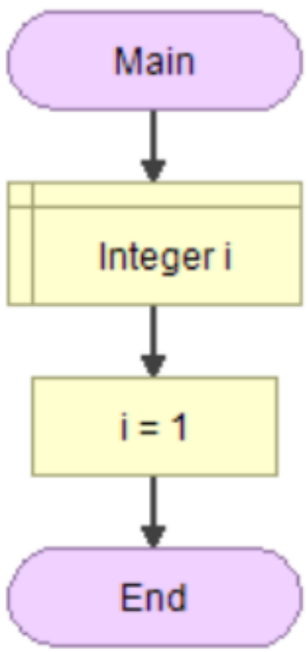
```
Enter x1: 2

Enter x2:3
3
```

- In general the two programs give different results.
- However, we may want to use one program for one set of `{x1,x2}` and the other program for another set of `{x1,x2}`.

.

Flow charting

- Flow charting is a general way of representing program flow.
- The basic idea is that each task or command is represented by a box and arrows connect one command to another.
- Flowgorithm converts flow charts into programs!
- We will use it to illustrate different types of program flow.

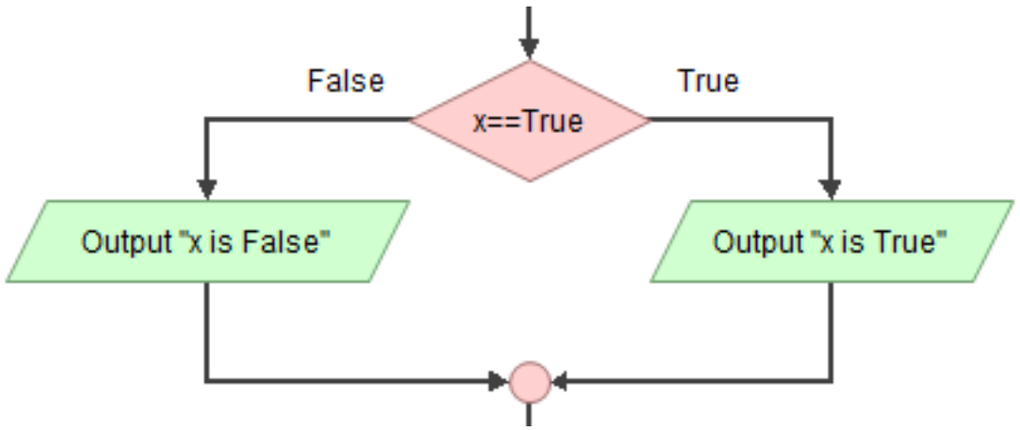


Other kinds of instructions

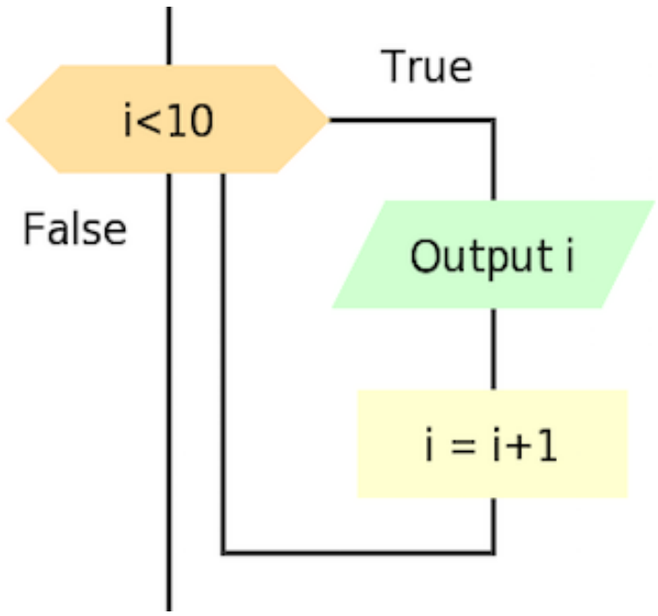
1. Making a decision
   - **If** a condition is satisfied, **then** do something
2. Repeat a task
   - Do something **for** a certain number of times
   - Do something **while** a condition is true

We will see many examples of these kinds of instructions.

Not all program flow is linear!



Making a decision



Repeating a task

2. Making decisions

Motivation

- Making decisions is one of the basic tasks performed by computer program.
- Without this capability, we would need to write a different program for each case. This is extremely inefficient!
- Making decisions introduces branches into the program flow.

Example 1: decision making with sequential flow

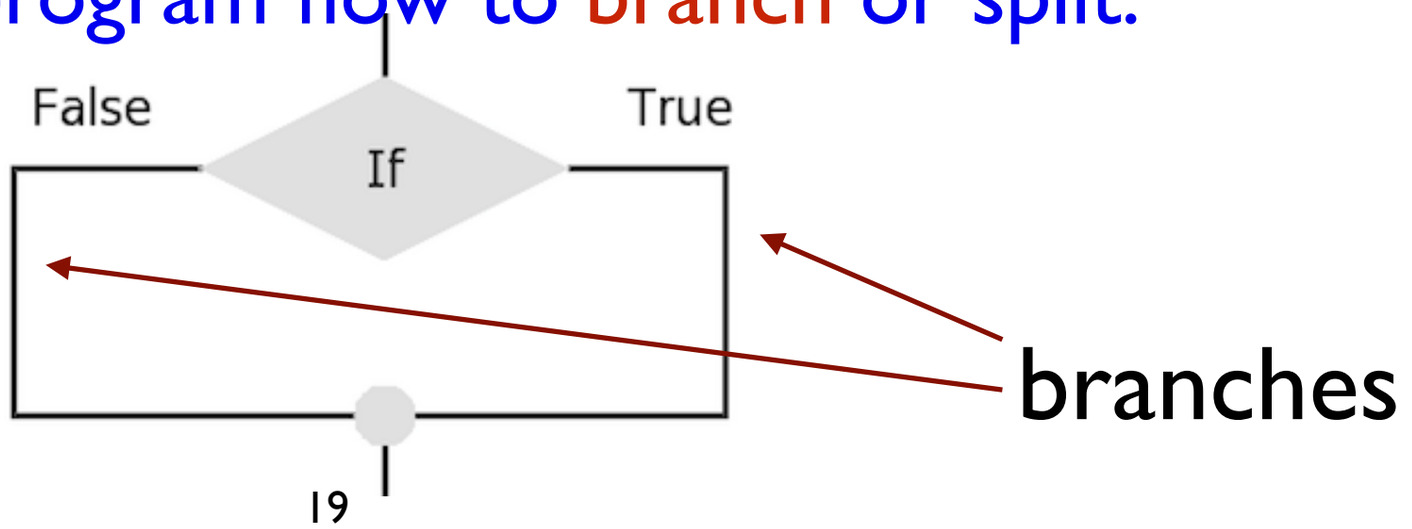
Can we write a program that calculates `0.5*x` for `x<0` and `2*x` for `x >0` using completely sequential flow?

```
x=float(input('enter a number: '))

# calculate 0.5x
# x < 0
# out = 0.5*x

# calculate 2*x
# x > 0
out = 2.0*x

print(out)
```

The only way to get the desired result is to comment out the code as required. This isn't practical!

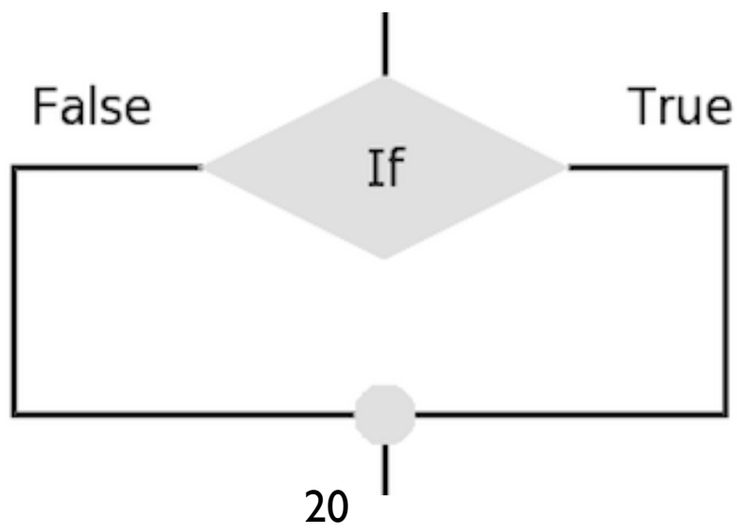
Making decisions manually

- In the previous example, we want to execute different commands depending on a condition (e.g. whether x is positive).
- This amounts to making a decision.
- In computer science, we make decision using an if test:
- In Flowgorithm, a decision is represented by the diamond .
- It causes the program flow to branch or split.



`if` test

- The basic building block of decision making in any computer language is the if test.
  - if a condition is `True`, carry out one set of instructions
  - if a condition is `False`, carry out another set of instructions.

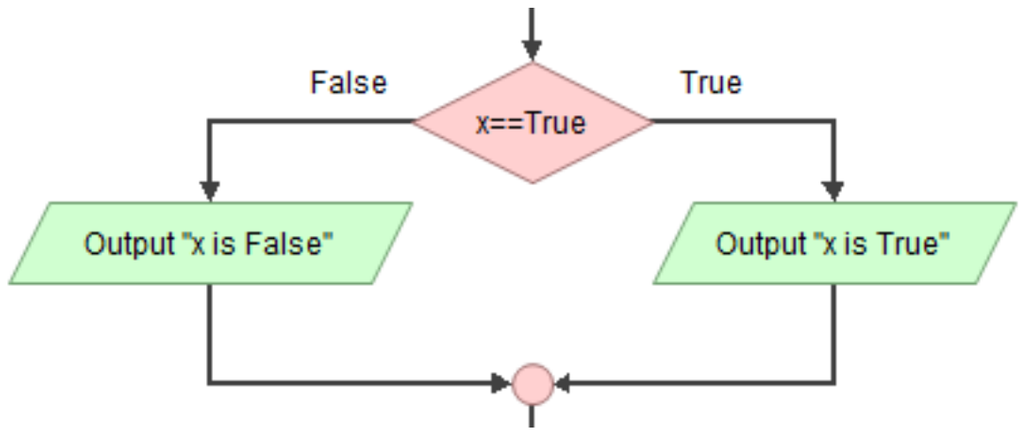


Implementation in Python

- Some version of the if test exists in any every computer language. Details vary slightly from language to language.
- This is what it looks like in Flowgorithm and Python:

condition is True →
condition is False →

```
x=False

if x==True:
    print('x is True')
else:
    print('x is False')
```



Syntax of the `if` statement

- More precisely, the `if` statement has the following syntax:

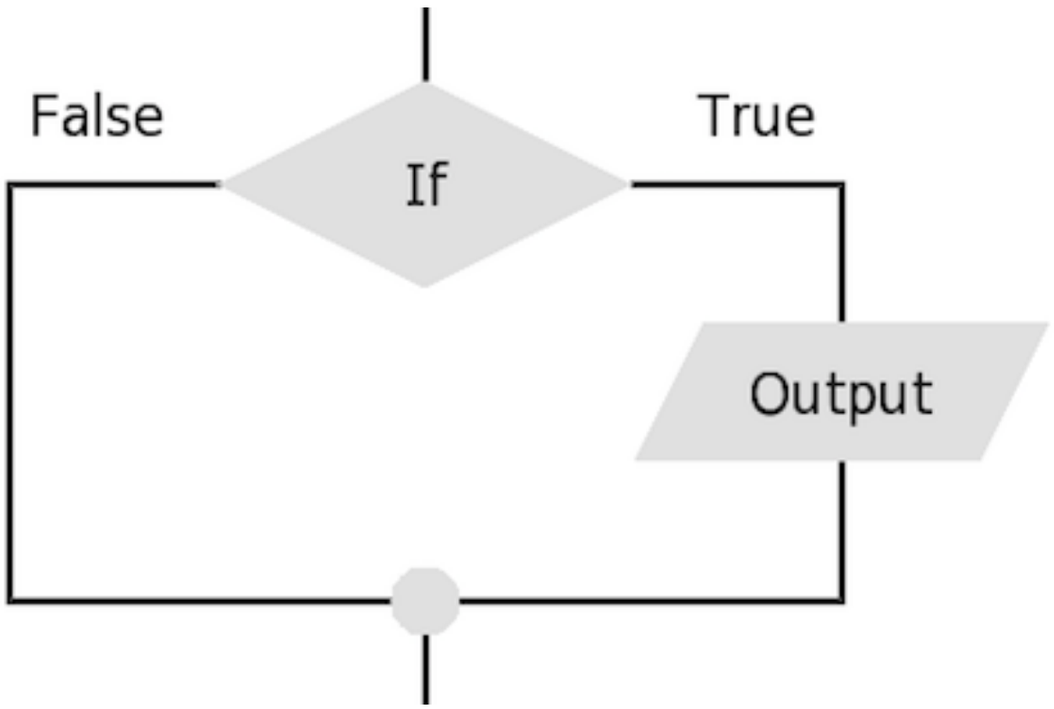
```
if <expression>:
        statement 1
        […]
else:
        statement 2
        […]
```

executed if *`expression`*`==True`

executed if *`expression`*`==False`

Short form

If one only wants to execute code when the expression is `True`, then the else block can be omitted.




```
x=True

if x==True:
    print('x is True')
```

Getting the syntax right

- It's important to remember that the syntax must be followed exactly. In particular the following rules need to be observed:

  1. Colon follows `if` and `else`
  2. The statements to be executed following `if` and `else` must be indented by a fixed number (typically 4) of spaces.

- If these rules are not followed, there will be an error:

Spyder has detected a problem

```
1
2
3  if x==True
4      print('x is True')
5  else:
6      print('x is False')
7
```

```
x=False

if x==True
        ^
SyntaxError: invalid syntax
```

Python reports a syntax error when we try to run the program

Whitespace in Python

One of the defining features of Python is its use of whitespace.

- **Multiple statements can be executed** sequentially by *indenting them by the same amount.*

```
if <expression>:
    statement 1
    […]
else:
    statement 2
    […]
```

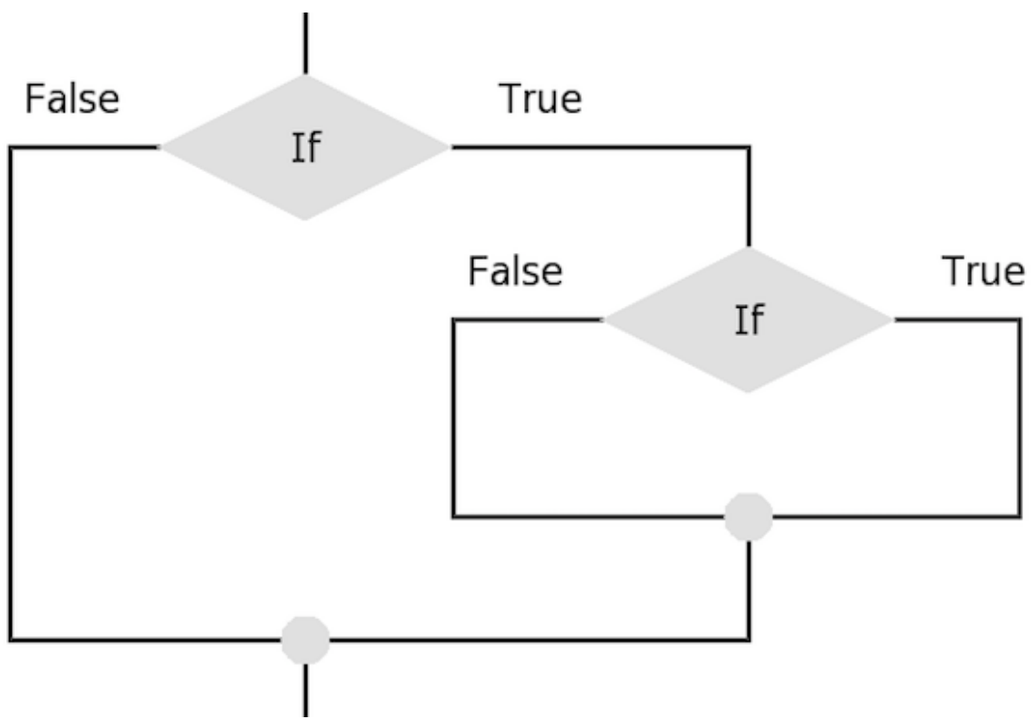
executed if *expression*==True

executed if *expression*==*False*

```
x=False

if x==True:
    print( 'x is True' )
    print( 'bye' )
else:
    print( 'x is False' )
    print( 'bye' )
```

Nested `if`s

- One can nest an `if` statement inside an existing one by indenting it.
- One can nest arbitrarily many times. But it's bad programming practice to go more than 3 levels deep.




```
x=True
y=False

if x==True:
    print('x is True')
    if y==True:
        print('y is True')
    else:
        print('y is False')
else:
    print('x is False')
```

General expressions

- For simplicity we've only considered simple tests (e.g. `x==True`). In practice, *any valid Python expression can be used.*

- Expressions can be constructed from the relational and logical operators discussed in the previous section.

Operator	Condition
<	Less Than
>	Greater Than
<=	Less Than or Equal to
>=	Greater Than or Equal to
==	Equal to
!=	Not Equal to

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

Fig. 2.7 The and operator

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

Fig. 2.8 The or operator

A	not A
False	True
True	False

Fig. 2.9 The not operator

Comparison of floats

Remember that floating-point numbers have limited precision. This is usually fine, but comparisons involving == can sometimes yield surprises.

```
x=2**0.5
if x**2.0 == 2.0:
    print('(x**0.5)^2 = 2')
else:
    print( 'We have a round-off error' )
    print('x**2=',x**2)
```

```
We have a round-off error
x**2= 2.0000000000000004
```

3. Extensions

List of alternatives

- The simple `if` test is incredibly important in computer programming.
- But it doesn't cover all possible cases. Sometimes we want to choose among a list of alternatives:
  - Choice A → Instruction set 1
  - Choice B → instruction set 2
  - Choice C → instruction set 3
  - etc.

Naive implementation

The following program chooses among a list of alternatives:

```
x=int(input('Enter an animal type: '))
if x==0:
    animal='dog'
else:
    if x==1:
        animal='cat'
    else:
       if x==2:
            animal='mouse'
       else:
            if x==3:
                animal='penguin'
            else:
                animal='missing'

print( 'The animal is: ' + animal )
```

This works but it's a bit clumsy: we have lots of extra `else` statements and the code is indented a lot. This makes it harder to read.

Use of `elif`

The same code can be rewritten more compactly using `elif`. It can be thought of as "else if", i.e., code to be executed if another condition is satisfied.

```
if <expression1>:
    [...]
elif <expression2>:
    [...]
elif <expression3>:
    [...]
else:
    [...]
```

We can have arbitrarily many `elif` blocks. But including too many makes the code hard to read!

Implementation with `elif`

We can rewrite the previous program using `elif`:

```
x=int(input('Enter an animal type: '))
if x==0:
    animal='dog'
elif x==1:
    animal='cat'
elif x==2:
    animal='mouse'
elif x==3:
    animal='penguin'
else:
    animal='missing'

print( 'The animal is: ' + ' ' + animal )
```

Note that this code is shorter and easy to read. This is important because the risk of error is reduced.

Error trapping

Occasionally we want to exit from a program when something happens. Approach:

1. Test for a condition
2. Exit

But how do we exit from our program?

Quitting a program

We can quit a program before we've reached the final statement. Unfortunately Python makes this a bit complicated:

1. Add `import sys` to the beginning of the program.
2. Exit by calling `sys.exit()`

```
import sys

variable='quit'

if variable=='quit':
    print ('selected quit')
    sys.exit()

print( "Didn't select quit!. now finished." )
```

```
selected quit
```

Summary

1. A program can be represented by a sequence of instructions or a flow chart.
2. A sequential flow of instructions is appropriate only for very simple programs.
3. Virtually all non-trivial programs require some form of decision making, which introduces branches into the program flow.
4. We make decisions in Python using `if-else,` which uses indents (whitespace) to group together statements.
5. Successive decisions can be made with `elif.`