Lecture 1

Introduction to Programming

1.

Why Learn Programming?

The Digital World

- People are spending increasing amounts of time in digital environments.
- These digital environments are defined by **software**:
  - “**Apps**” (applications), “**websites**”, “user interfaces”, “databases”, “artificial intelligence”, etc.
- The **software** that we interact with *shapes our lives*.
  - Social media is a prime example.

Learning how software works empowers you, and opens up new opportunities in an ever-expanding digital world.

Computers are Everywhere

- To appreciate how much **software** you **interact** with, think of all of the devices you use:
  - Laptops, phones, TVs, self-service checkouts, etc.

- Sometimes **software** is mostly **unseen**:
  - EFTPOS, washing machines, cars, etc.

- **Programming is the practice of creating software**.
  - *You are learning how to control the most important machines on the planet!*

Programming Saves You Time

- Have you ever found yourself in a situation where you are performing repetitive tasks on a computer?

    - Renaming a bunch of songs one-by-one?
    - Cropping a whole album of photos?
    - Downloading many files individually?
    - Sifting through documents to pull out information?

Programming Saves You Time

- You can write your own computer programs to perform time-consuming tasks.
- It’s like being able to create your own little digital robot assistant.
  - *This only becomes more useful as the digital world expands.*

Anecdote: Car Audio

- I have a large digital music collection.
- I also have a fairly old car, so I installed a stereo which can play music from USB.

- When I tried to use it, I ran into a few issues:
  - The track order within albums was wrong.
  - The volume across songs was inconsistent.
  - Some songs just wouldn’t play at all.

But… but…

…

…

…

...I just wanted to jam in my car... 😥

Doing Things Manually == Sadness

- I could have manually gone through track-by-track, using audio editing software to convert every song into MP3 format and adjust the volume and fix up metadata and so on...
- Sure, this would work, but it would take me hours.
- Furthermore, whenever I wanted to add fresh music I'd have to do it all again!

Automation for a Happier Existence

- Instead, I used my programming knowledge to write a small amount of code (software).

- This code was able to convert the songs to MP3, normalise the volume, copy them to USB, and sort them correctly.

- After a few minutes, I was able to jam in my car! 

Career Opportunities

- Computers are used in almost all industries.
  - Medicine, agriculture, engineering, etc.

- As a result, **programmers** are in **high demand**.

- Job opportunities are varied
  - *Website design*
  - *Data analysis*
  - *Research*
  - ...

Programming is Fun and Rewarding

- Besides everything else, programming can just be a fun hobby.
- Finding a good way of solving a *tricky programming* task can be very satisfying.
- You can turn an idea into a working piece of **software** and *share it with the world*.
  - Got an idea for a cool video game? Learn to program and make it!

Computational Thinking

Human Thinking

- As humans, we have common sense and are able to "read between the lines".
- We are good at reasoning and using our past experiences and intuition to solve problems.
- Humans are **creative** and **independent** thinkers.

Computer Thinking

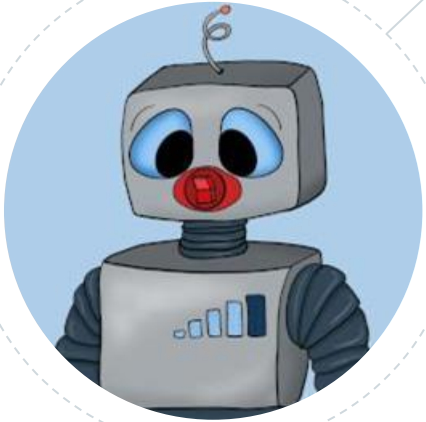
- On the other hand, computers are extremely logical and do exactly what they are told.
- Computers do not “read between the lines” and can’t “figure things out for themselves”.
- Computers are **literal** and **subordinate** thinkers.
- This may seem frustrating at times, but computers make up for it in other ways.
  - For example, modern computers can perform billions of calculations per second!

The "Baby Robot" Analogy

- A computer is like a baby robot.
  - No prior experience.
  - Does exactly what you tell it.

- You need to give clear, unambiguous, step-by-step instructions.

- A set of such **instructions** is called an **algorithm**.



What is an Algorithm?

- An **algorithm** is a well-defined **sequence** of step-by-step **instructions**.

- There is *no single algorithm*---different tasks require **different** algorithms.

- An algorithm is a lot like a **cooking recipe**.
  - Individual, detailed steps with a defined order.

- **Programmers design *algorithms* and turn them into** computer **programs**.

An Algorithm for Making a Sandwich

1. Place two slices of bread on the bench.
2. Spread butter on one slice of bread using a knife.
3. Place one slice of ham on the buttered slice.
4. Place two pieces of lettuce on top of the ham.
5. Place the unbuttered slice of bread on top of the lettuce.

Bugs

- A **bug** is a flaw in **software (program)** which results in incorrect behaviour.
- Bugs can arise due to logical errors in an algorithm.
- It is important to think algorithms through carefully to avoid introducing bugs.
- The process of finding and fixing bugs is called **debugging**.

Python

What is a Programming Language?

- Computers are logical machines that **follow** instructions very literally.

- Computers **do not understand** plain English.

- In order to **describe** an **algorithm** to a computer, it is necessary to **write** a **computer program** using a **programming language**.
  - **This is also known as "coding".**

What is a Programming Language?

- A **programming language** is a structured, unambiguous way of describing an algorithm.

- Programming languages have a **strict syntax**.
  - Much **stricter** than grammar in **natural language**!
  - "**Typos**" such as a **misplaced comma** or **misspelled word** can **prevent** an entire **program** from working.

Machine Code

- The fundamental language understood by computers is **machine code**.

- Machine code is fast and natively understood by computers, but is **very, very difficult** for humans to **write directly**.

Other Programming Languages

- Writing a program in machine code is like baking a cake with a chicken, cow, and some wheat---possible, but unnecessarily time-consuming.
- Other programming languages have been **developed** which ***translate*** into machine code.
- This **allow** humans to indirectly **produce** machine code in a much easier way.
- **Python** is one such **programming language**.

Python

- The first version of the Python programming language was released in 1991.
- Python places a large emphasis on **code readability**.
  - That is, Python is designed to be **human-friendly**.
- The language has improved over the years.
- Our code will work with Python version 3.6 (released in 2016) and newer.

Python is Readable

Even with no programming experience, you should have some idea about what this Python code does:

```
for item in store_inventory:
    if item.stock <= 2:
        print('Low stock for ' + item.name)
```

"For each item in the store's inventory, if there are 2 or fewer in stock, display a low stock message with the item name."

Getting Python

- Python is free for everyone and can be installed on Windows, Mac, and Linux computers.

- You can download and install Python here:
  - https://www.python.org/downloads/

- There are instructional videos on LMS which walk you through setting up a Python development environment on your own computer.

2.

Introduction to Flowcharts

Motivation

- Let's say you want to start a housecleaning business.
- You hire three employees and start taking customers.
- After running your business for a while, you start receiving complaints about inconsistent results.
- You decide to investigate:
  - You take each employee to a bedroom, and instruct them to “clean the room”.
  - You observe different processes!

What Went Wrong?

- “Clean the room” is not a precise description of the task to be performed.
- Human beings fill in the gaps based on “human intuition”:
  - Past experiences,
  - Preferences,
  - Mood,
  - etc.

What Can Be Done?

- Document the steps in detail, without room for interpretation.
  - i.e. **design the algorithm**.

- Hand out the detailed instructions.

- Each employee can now follow the exact same process.

Motivation

- Describing processes in detail is especially important for machines.
- Computers typically do exactly what they are told and lack any “human intuition”.
  - *Recall the baby robot analogy*.
- This means that a **detailed algorithm** is not just **preferable---*it is required***.

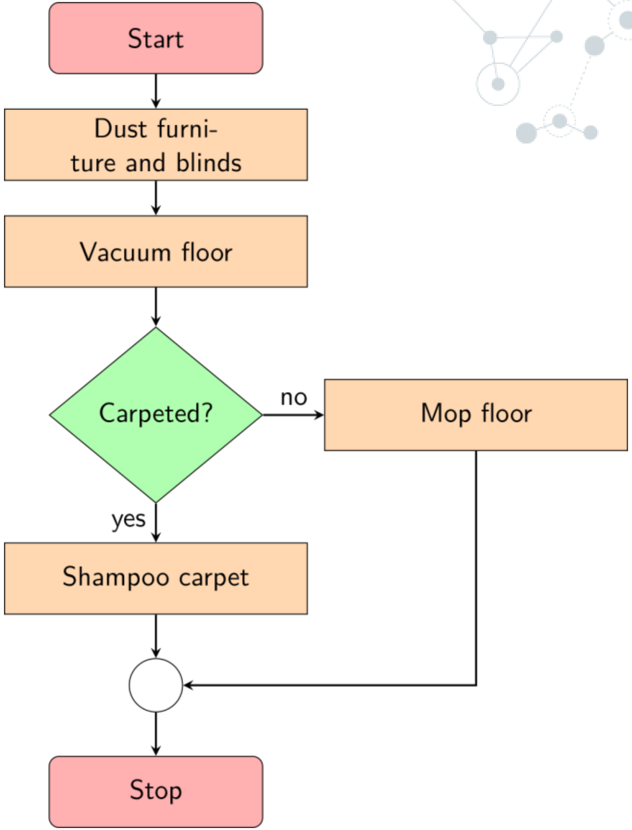
Describing Processes

- One way of describing a process is through **written paragraphs**.
- **Complicated** processes can be **difficult** to read.

Begin by dusting the furniture and blinds. Next, vacuum the floor. If the floor is carpeted, then you should shampoo the carpet. Otherwise, if the floor is not carpeted, mop the floor.

Describing Processes

- Another way of describing a process is with a **flowchart**.
- A **flowchart** provides a clear visual representation.



What is a Flowchart?

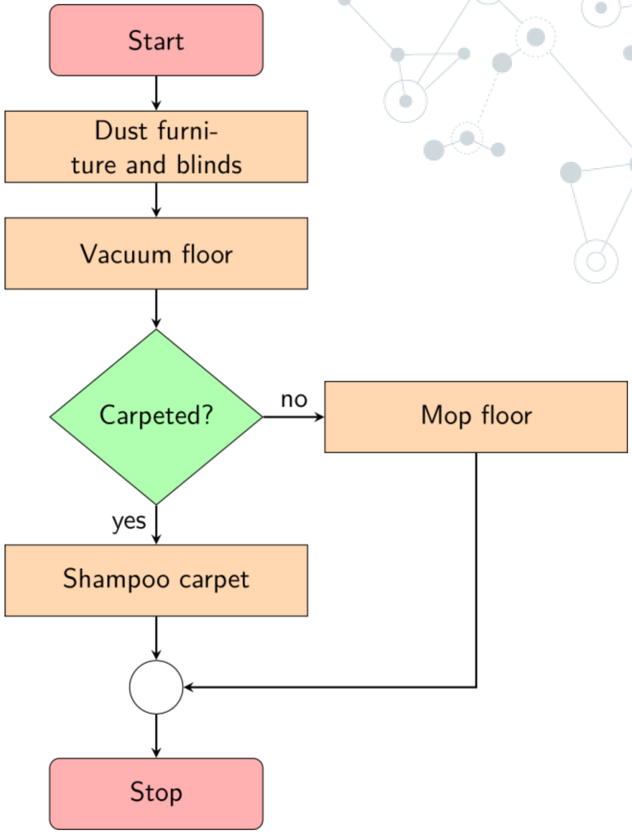
- A **flowchart diagram** details the flow of a process.
  - Used to document or communicate a process.
  - Describes steps clearly and unambiguously.
  - Applicable to both physical processes and computer processes.

What is a Flowchart?

- **Flowcharts** can be a useful tool for **designing** computer **programs** *before* **writing code**.
    - Helps you to think through the **individual** decisions and **steps** involved in a process.
    - Can be translated into actual **programming code** afterwards.

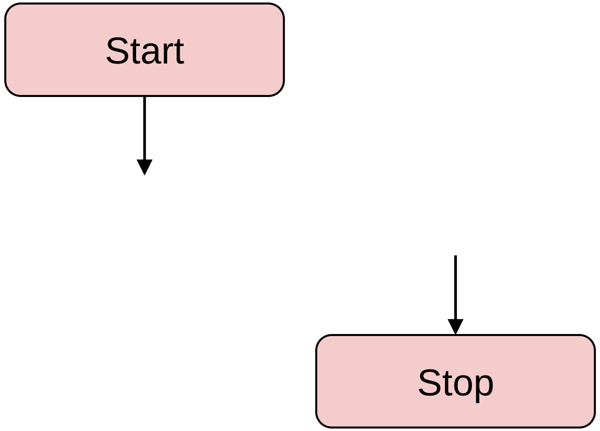
Elements of a Flowchart

- A **flowchart** consists of **shapes joined** with **arrows**.
    - The **arrows** describe the **flow** of the process.
    - The **shapes describe** *steps* in the *process*.
- We will now describe the meaning of each of the different **shapes**.



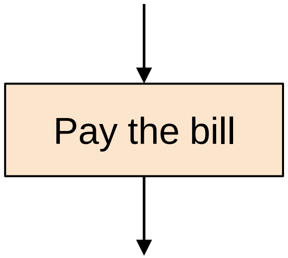
Start/Stop Elements

- Start/stop elements describe points where a process begins and ends.
- Start element: 1 outgoing arrow.
- Stop element: 1 incoming arrow.
- Each flowchart should include one start element and one stop element.
- Represented using rounded rectangles.



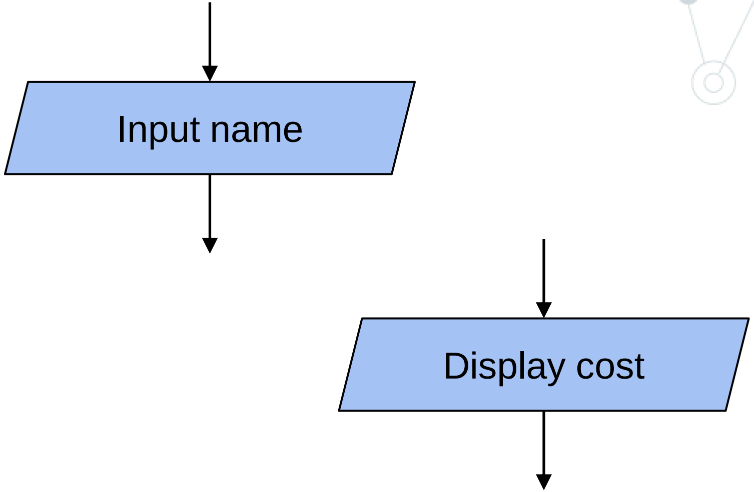
Process Elements

- Process elements describe a processing step.
- 1 incoming arrow and 1 outgoing arrow.
- Description usually starts with a verb (action).
- Flowcharts can include many process elements.
- Represented using rectangles.



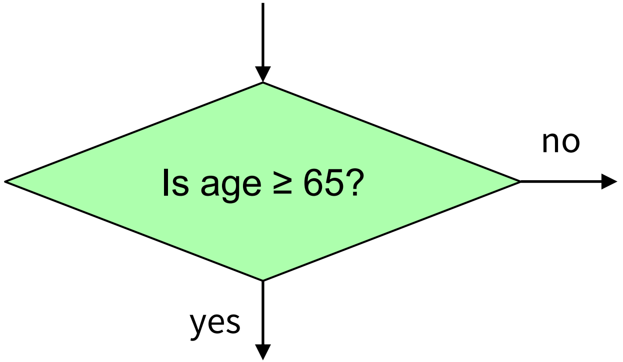
Input/Output Elements

- Input/output elements describe points where data enters/leaves a program.
  - e.g. Accepting user input, displaying results.
- 1 incoming arrow and 1 outgoing arrow.
- Flowcharts typically include at least one input and one output element.
- Represented using parallelograms.



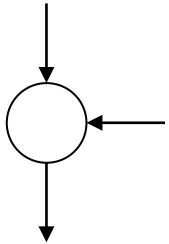
Decision Elements

- Decision elements select between multiple flows based on some kind of test condition.
- 1 incoming arrow and multiple outgoing arrows.
- Often phrased as a yes/no question.
- Represented using diamonds.



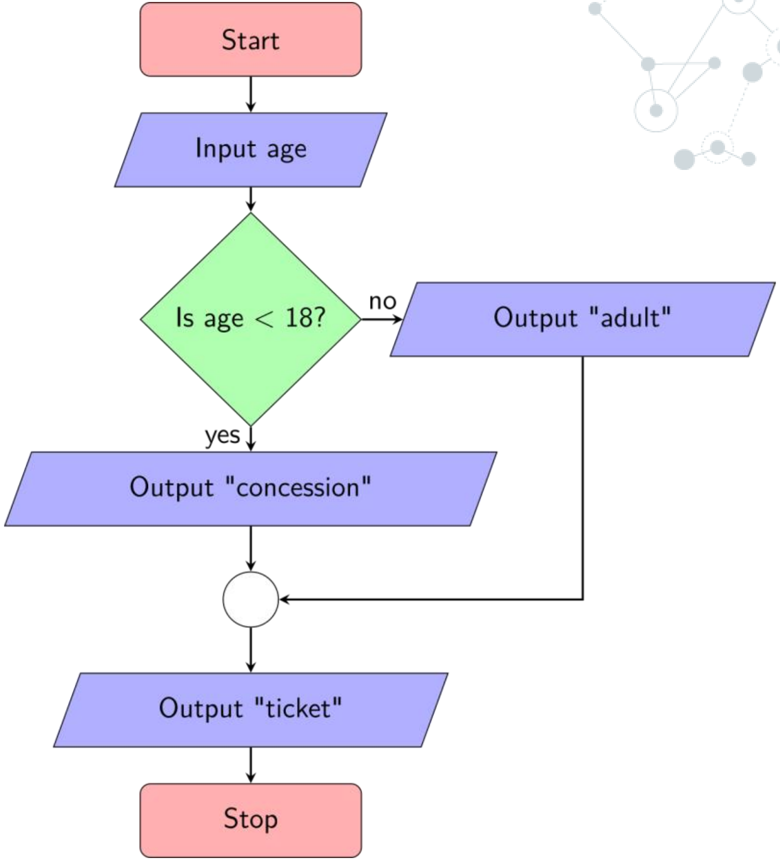
Connectors

- Connector elements join multiple flows.
- Multiple incoming arrows and 1 outgoing arrow.
- Useful for rejoining multiple flows after a decision.
- Represented using circles.



Check Your Understanding

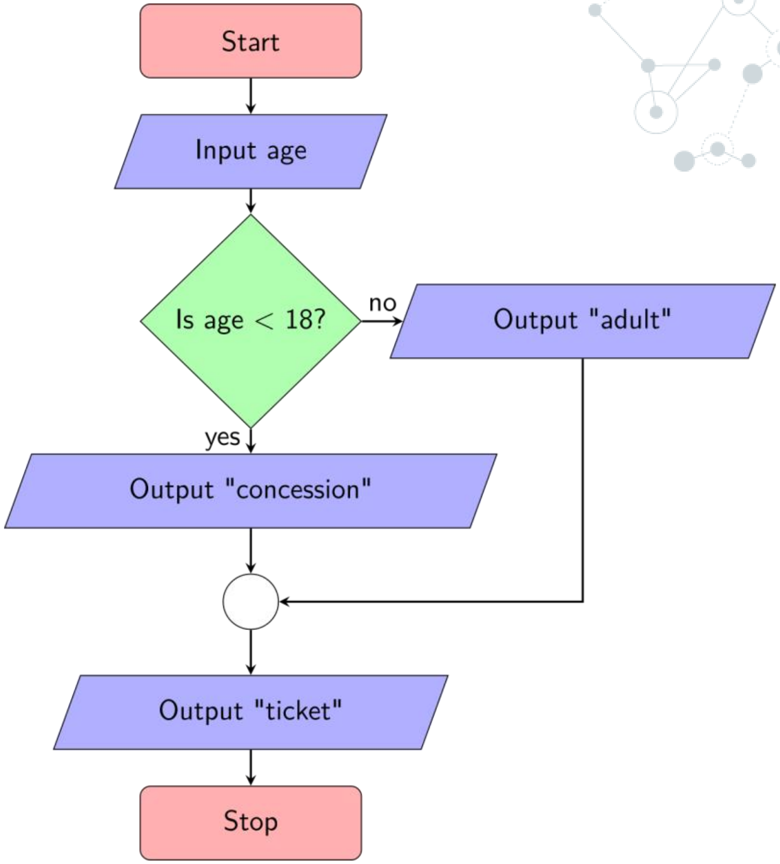
Q. What are the expected outputs of the program described by this flowchart when the user inputs an age of 19?



Check Your Understanding

Q. What are the expected outputs of the program described by this flowchart when the user inputs an age of 19?

A. "adult" and "ticket". "adult" is output because age (19) is not less than 18. "ticket" is always output irrespective of the input age.



Example Flowchart: Guessing Game

Task Definition

The computer picks a number between 1 and 100, and the user tries to guess that number. Whenever the user makes an incorrect guess, they are told whether the number is higher or lower than their guess. When the user correctly guesses the number, a victory message is displayed and the program stops.

Example Behaviour

1. Generate a random number (**x** ← 44).
2. Input user guess (**y** ← 25).
3. Is **y** equal to **x**? No.
4. Is **x** greater than **y**? **Yes**, display "Higher".
5. Input user guess (**y** ← 50).
6. Is **y** equal to **x**? No.
7. Is **x** greater than **y**? **No**, display "Lower".
8. Input user guess (**y** ← 44).
9. Is **y** equal to **x**? **Yes**, display "Correct".

Identifying Flowchart Elements

- We can analyse the example behaviour to identify flowchart elements:

  - **Input/output**
  - **Decision**
  - **Process**

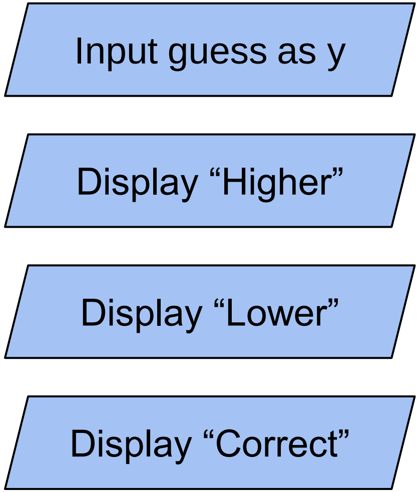
Identifying **Input/Output** Elements

1. Generate a random number (**x** ← 44).
2. Input user guess (**y** ← 25).
3. Is **y** equal to **x**? No.
4. Is **x** greater than **y**? Yes, display “Higher”.
5. Input user guess (**y** ← 50).
6. Is **y** equal to **x**? No.
7. Is **x** greater than **y**? No, display “Lower”.
8. Input user guess (**y** ← 44).
9. Is **y** equal to **x**? Yes, display “Correct”.

Identifying Input/Output Elements

- Inputs:
  - User guess (a number, y).

- Possible outputs:
  - Higher hint (“Higher”).
  - Lower hint (“Lower”).
  - Victory message (“Correct”).

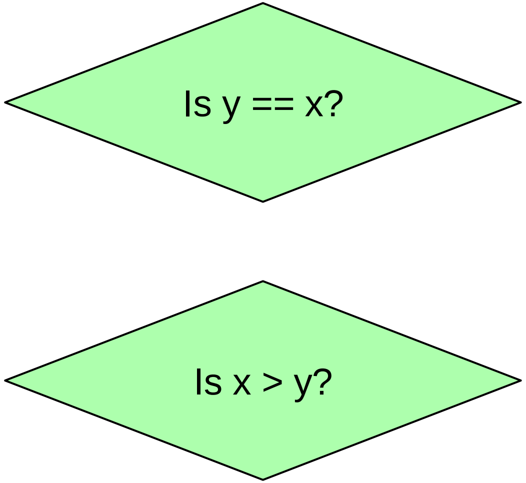


Identifying **Decision** Elements

1. Generate a random number (x ← 44).
2. Input user guess (y ← 25).
3. Is y equal to x? No.
4. Is x greater than y? Yes, display “Higher”.
5. Input user guess (y ← 50).
6. Is y equal to x? No.
7. Is x greater than y? No, display “Lower”.
8. Input user guess (y ← 44).
9. Is y equal to x? Yes, display “Correct”.

Identifying Decision Elements

- Is **y** equal to **x**?
- Is **x** greater than **y**?



Identifying **Processing** Elements

1. Generate a random number (x ← 44).
2. Input user guess (y ← 25).
3. Is y equal to x? No.
4. Is x greater than y? Yes, display “Higher”.
5. Input user guess (y ← 50).
6. Is y equal to x? No.
7. Is x greater than y? No, display “Lower”.
8. Input user guess (y ← 44).
9. Is y equal to x? Yes, display “Correct”.

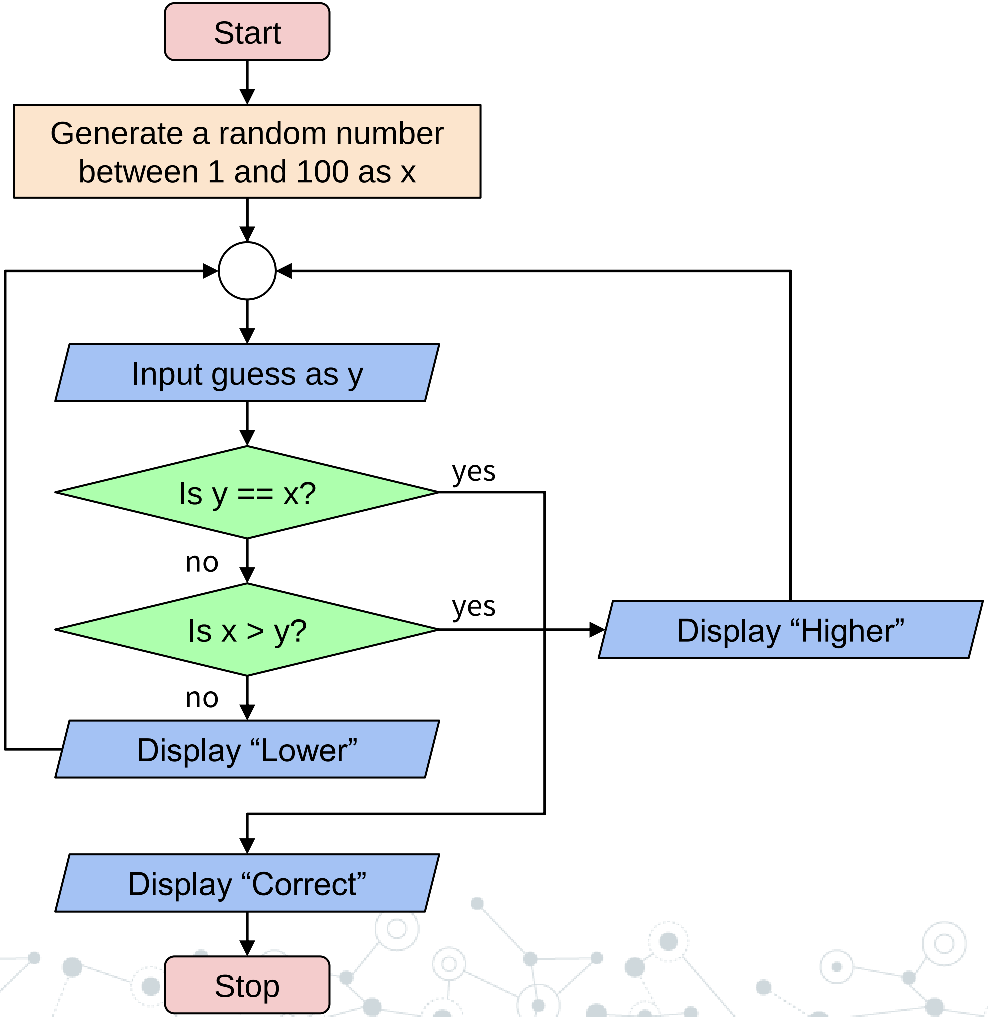
Identifying **Processing** Elements

- Generate a random number (x).

Generate a random number between 1 and 100 as x

Constructing the Flowchart

- Now that we have the elements, we can construct the flowchart.
- Need to think about the **flow** (**order of steps**).
- Once again, the **example behaviour** is a good reference.


Start
Generate a random number between 1 and 100 as x
Input guess as y
Is y == x?
yes
no
Is x > y?
yes
Display “Higher”
no
Display “Lower”
Display “Correct”
Stop

Next Steps

- Computers **can’t read flowcharts** like this one directly.
- The **next step** towards **creating** a **working program** would be to **translate the flowchart into code (program)**.
    - *This will be covered in **future** lectures.*
- Having a **well-designed** flowchart makes **writing** code much **easier**.

Designing Effective Flowcharts

Order is Important

- Order is important.
- Think carefully about the order in which things should happen as you draw the flowchart.

Would you trust this recipe?

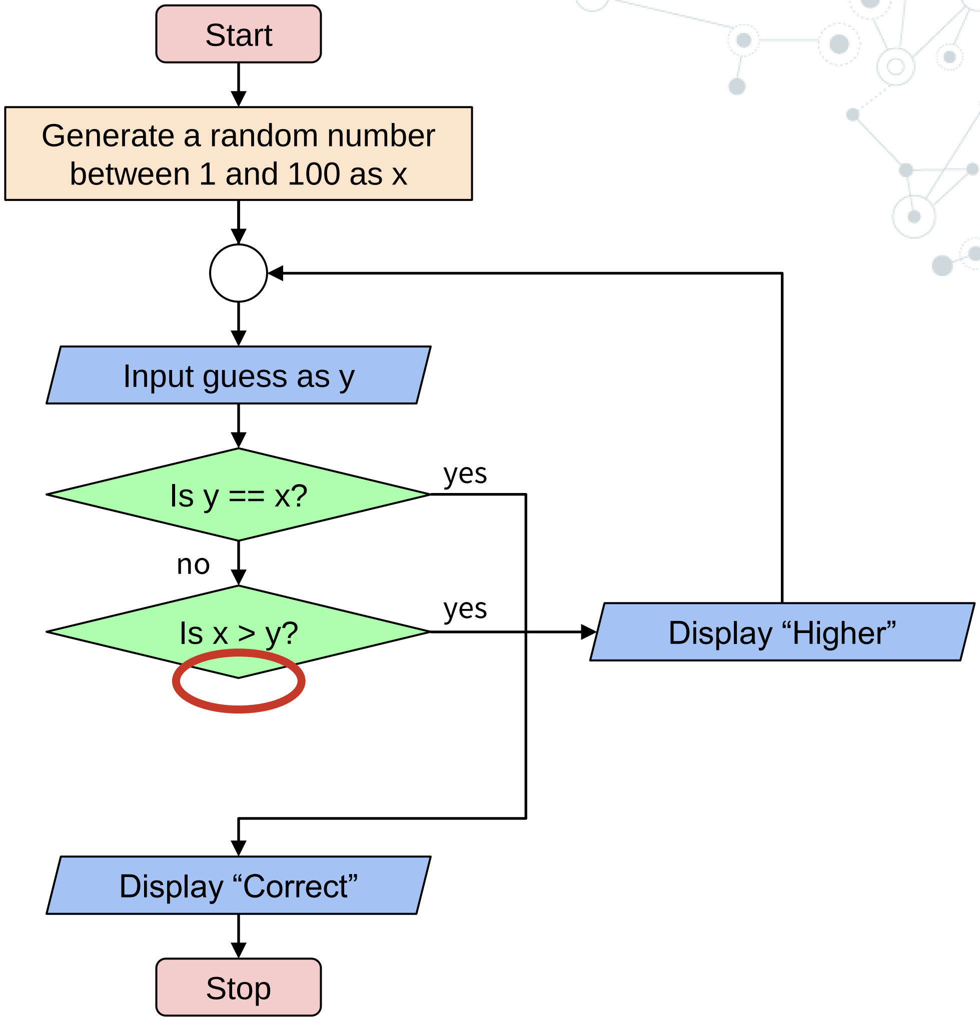
1. Take the cake out of the oven.
2. Mix ingredients thoroughly.
3. Allow the cake to cool.
4. Put the cake into the oven.
5. Add ingredients into the bowl.

Clear, Concise, and Complete

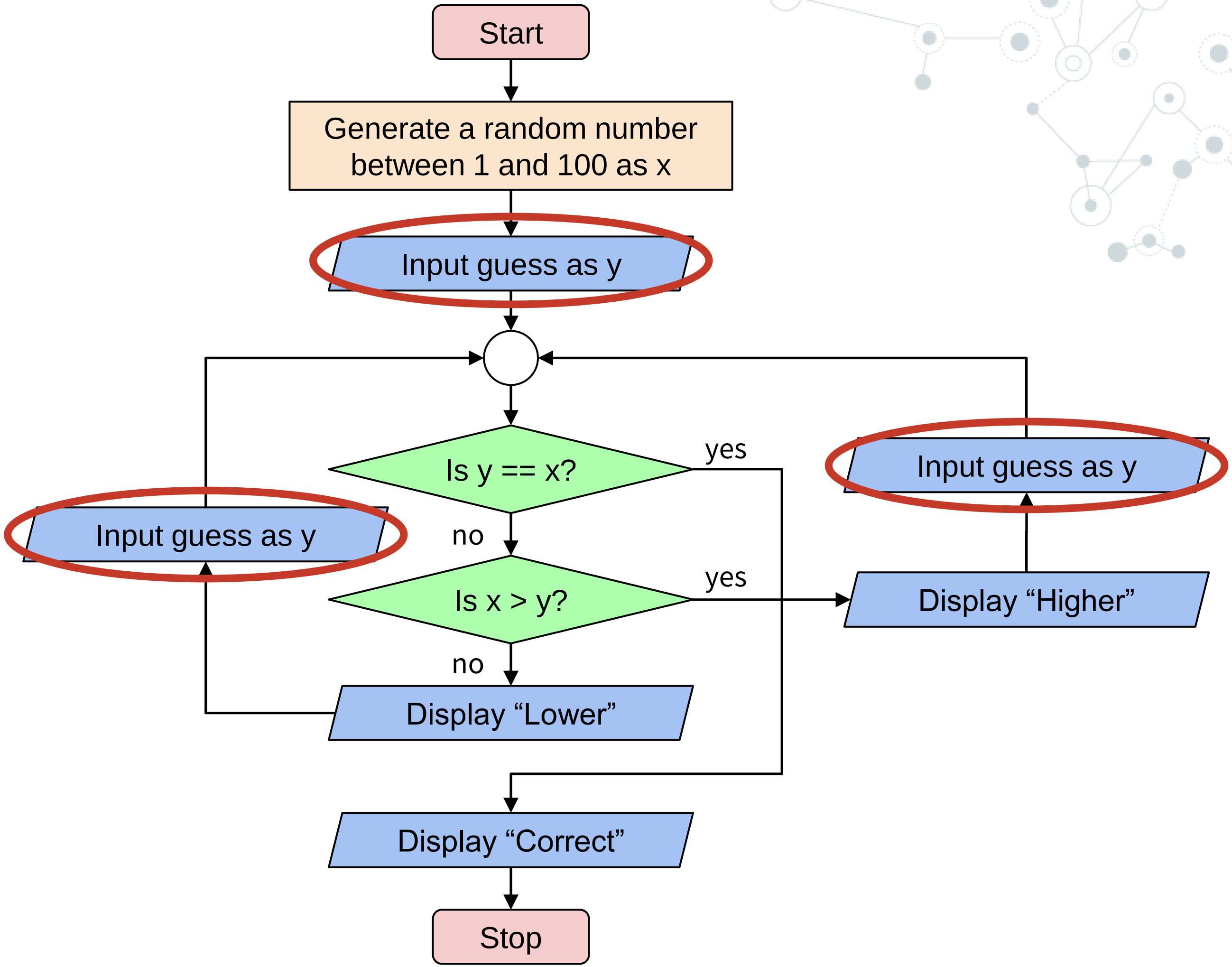
- Clear
  - Space elements out.
  - Avoid crossing over arrows where possible.

- Concise
  - Use short, accurate descriptions.
  - Avoid unnecessary duplication.

- Complete
  - Cover all possible eventualities.



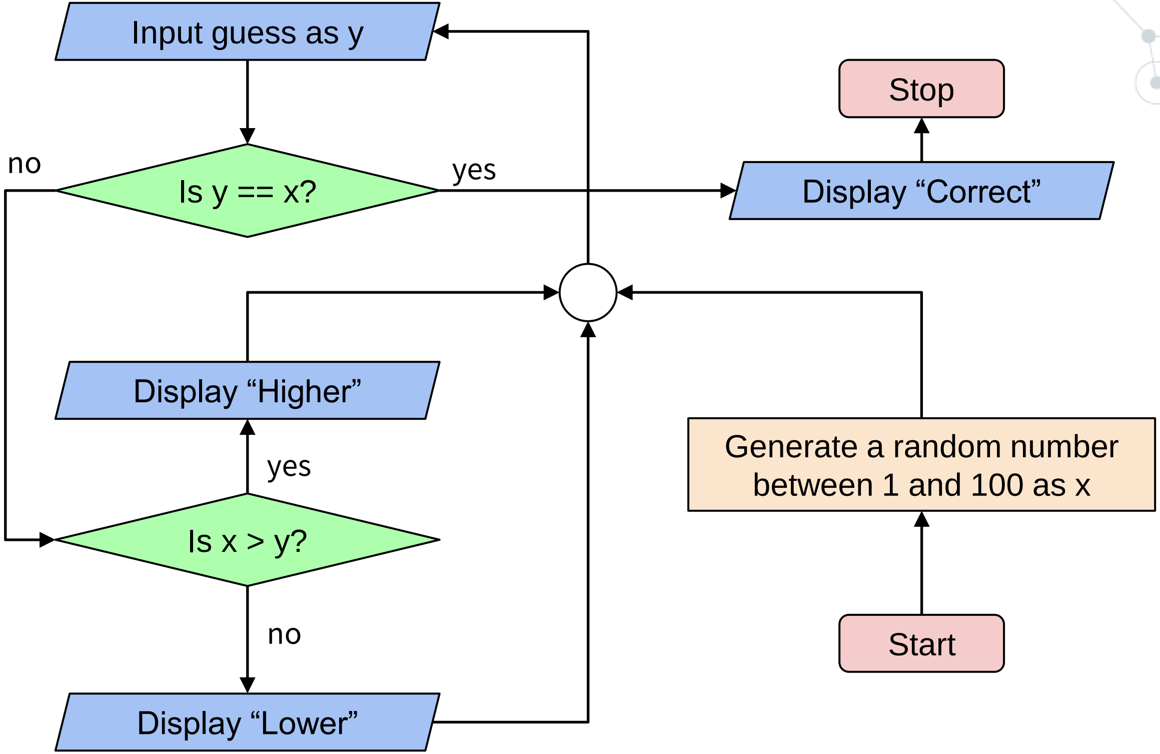
Incomplete. What happens if x is not greater than y?



Unnecessarily repeated elements add clutter.

Top-To-Bottom Layout

- Aim for the process to flow from top to bottom.
- Arrows should generally point downwards.
- Put the start element at the top of the page.
- Put the stop element at the bottom of the page.
- Sometimes decisions cause the flow to go back up, this is OK.



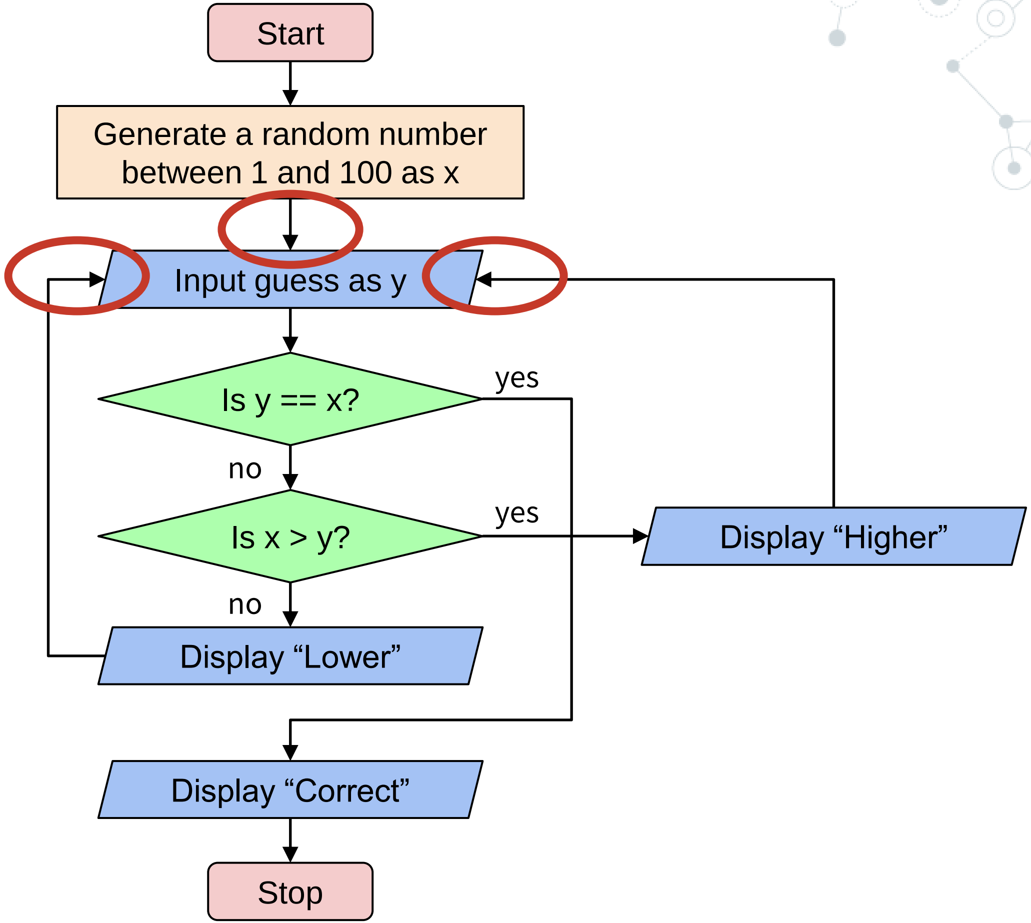
Inconsistent flow direction is hard to follow.

Know Your Arrows

- Different elements have different numbers of incoming and outgoing arrows.
- Check that each element in your flowchart has the correct number of arrows.
- Use connectors where appropriate.

ELEMENT	INCOMING ARROWS	OUTGOING ARROWS
Start	**0**	**1**
Stop	**1**	**0**
Process	**1**	**1**
Input/Output	**1**	**1**
Decision	**1**	**2+**
Connector	**2+**	**1**

A table showing the correct number of arrows for each flowchart element.



Incorrect number of incoming arrows. Use connectors to make joined flows obvious.

Flowchart Creation Checklist

✓ Is there one start element and one stop element?

✓ Are the elements in a logical order?

✓ Is the flowchart clear, concise, and complete?

✓ Does the flowchart flow from top to bottom?

✓ Is the number of arrows correct?

3 Literals and Variables

Users vs. Programmers

- **Users** see computers as a set of **tools** - word processor, email, excel, note, website, messenger, etc.

- Programmers **learn** computer **languages** to write a Program.

- Programmers use some **tools** (Python) that allow them to build new tools (**Program** or **Software**).

- Programmers often **build** tools for lots of **users** and/or for **themselves**.

What is a program?

What is a program?

Program

Program is a set of actions (or rules) to accomplish a specific task.

What is a programming language?

Programming language

A programming language comprises a set of instructions to produce various kinds of output. Programming languages are used in computer programming to implement algorithms.

Examples of computer programming languages are:

- **Python**
- C, C++
- JAVA

A Program for Humans...

Program

Shopping, sports, study, .. etc.

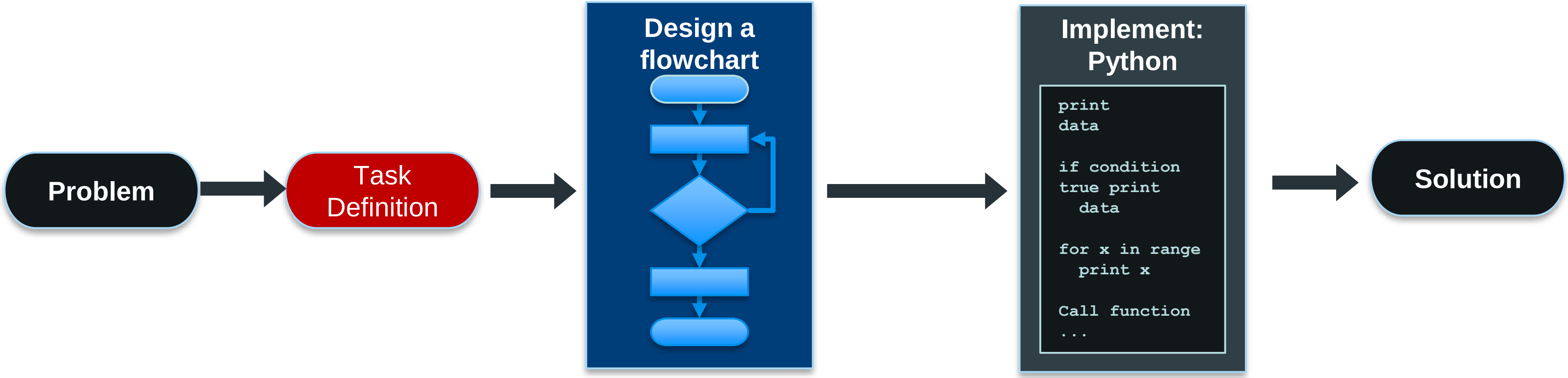
A Program for Python...

```python
class test:
    def __init__(self):
        self.password = ""

    def enter_password(self):
        try:
            self.password = input("Please enter password> ")
        except ValueError:
            print("Oops! That was not the correct password. Try again...")
        else:
            print("Legal input")
        return self.password
```

Writing a program

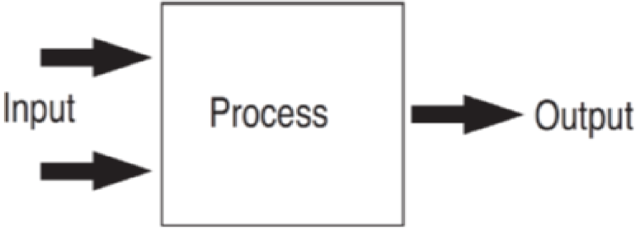
- A program can be used to *solve complex problem*
  - Programs are a set of written actions in order to fulfil a need / solve a problem
  - A programming language is the tool used to create a solution (Program)



By writing a program using Python, you can make a computer do something useful

Computer Program

A program is a ***sequence of instructions that specifies how to perform a computation***.



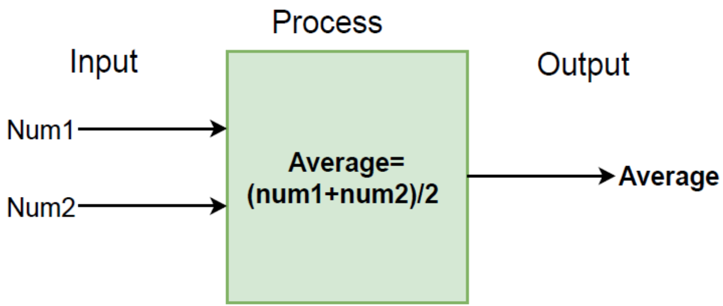
- **Input**: Get **data** from the **keyboard**, a **file**, the **network**, or some other **devices**.
- **Process:**
  - **Math**: Perform basic **mathematical operations** like **addition** (+) and **multiplication** (*).
  - **Conditional execution**: Check for certain **conditions** and run the appropriate **code**.
  - **Repetition:** Perform some action repeatedly, usually with some variation.

Output: Display data on the **screen**, save it in a **file**, send it over the **network**, etc.

Computer programs

Input, Processing, and Output- example

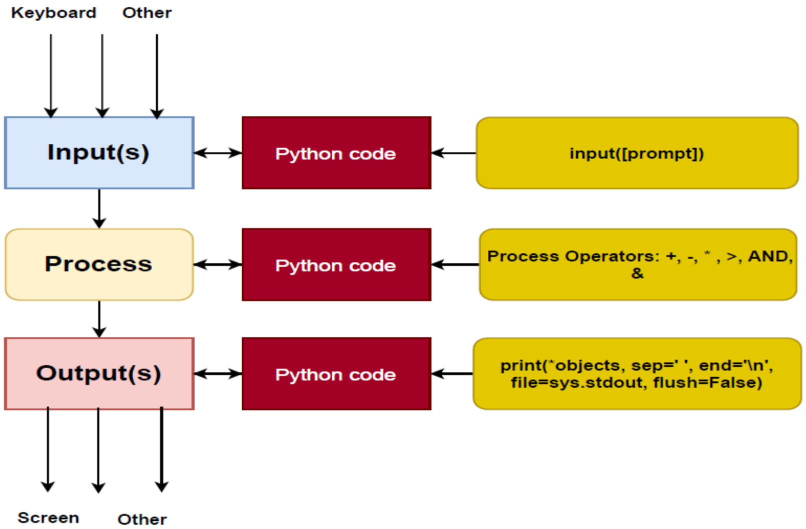
Example: Calculate the average of two numbers: num1 and num2.



Computer programs

Input, Processing, and Output

Python coding steps



Literals and Variables

Values

A value (Literal) is one of the basic things a program works with, like a number or a letter.

Addition

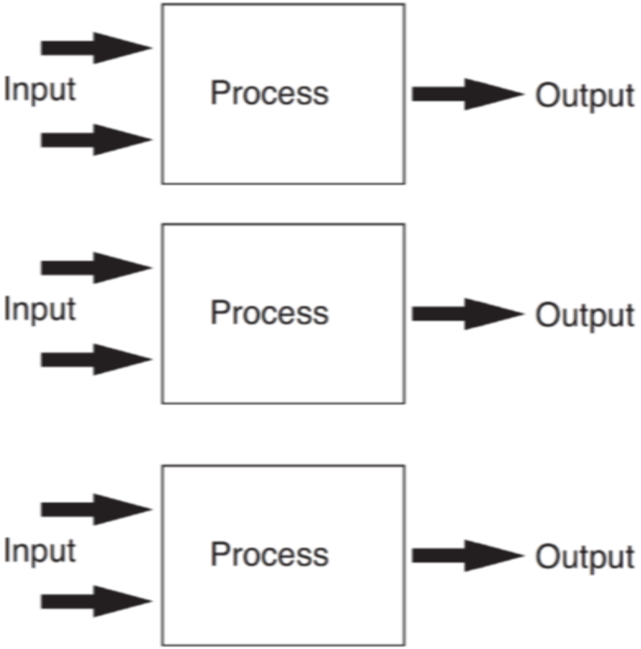
```
>>> 4 + 4
8
```

Subtraction

```
>>> 8 - 5
3
```

Multiplication

```
>>> 4 * 4
16
```



Literals

- **Literals** are values that Python interprets literally (as written).

- Represent **numbers**, **text**, and other **values** that are:
  - **Required** by the program, and
  - Known **ahead** of time.

- For example, a **calendar** program might contain **string** literals such as `'Wednesday'` and `'August'`.

Numeric and String Literals

- **Numeric** literals are written as you'd expect.
  - e.g. `42` means the number 42.

- **String** literals (snippets of text) are written between matched quotation marks.
  - e.g. `'text'` or `"text"`.

```
>>> 42
42
>>> 27.5
27.5
>>> 'Hello there!'
'Hello there!'
>>> "Double quotes work too."
'Double quotes work too.'
```

Check Your Understanding

Q. Which lines in the shown program contain numeric/string literals?

```
1 x = 5.1
2 y = x + 2
3 z = x + y
4 print('Result:')
5 print(z)
```

Check Your Understanding

Q. Which lines in the shown program contain numeric/string literals?

A. Lines 1, 2, and 4.

- Line 1: `5.1` (numeric).
- Line 2: `2` (numeric).
- Line 4: `'Result:'` (string).

```
1 x = 5.1
2 y = x + 2
3 z = x + y
4 print('Result:')
5 print(z)
```

Variables

- A variable allows you to give a name to a **value**.
- You can think of a variable like a labelled box.
- A variable is a **named** place in the memory where a programmer can store data and later retrieve the data using the variables “**name**”
- Variable **names** are **decided** by the programmer.
- The **contents** of a variable can **change**.

Creating Variables

- A variable can be defined (created) via assignment.
- **Assignment** links a variable name with a value.
- Assignment in Python is performed using the **equals** symbol, e.g. `x = 5`.

```
>>> day = 'Wednesday'
>>> day
'Wednesday'
>>> age = 28
>>> age
28
```

Single Equals (=)

Mathematics

- '=' indicates equality.
- An expression of state (how things are).
- x = 5 means "x is equal to 5".

$$x = 5$$

Python

- '=' indicates assignment.
- An instruction.
- x = 5 means "assign 5 to x" (i.e. "store 5 in x").

```
x = 5
```

Updating Variables

- Assignment can also be used to **update** the value of an existing variable.

- In the example shown, **x** originally has a value of **5** but is later **updated** to contain **10** instead.

```
>>> x = 5
>>> x
5
>>> y = 'Hello'
>>> y
'Hello'
>>> x = 10
>>> x
10
```

Updating Variables

Variable	Value

```
x = 5
y = 'Hello'
x = 10
```

Updating Variables

Variable	Value
x	5

```
x = 5
y = 'Hello'
x = 10
```

Updating Variables

Variable	Value
x	5
y	'Hello'

```
x = 5
y = 'Hello'
x = 10
```

Updating Variables

Variable	Value
x	~~5~~ 10.1
y	'Hello'

```
x = 5
y = 'Hello'
x = 10.1
```

Variable Naming Rules

- Variable **names**:
  - **Must** only contain **letters**, **digits**, and **underscores**.
  - **Must not** begin with a **digit**.

- Lowercase/uppercase matters.
  - `myVariable`, `MyVariable`, and `MYVARIABLE` are all **different** in Python.

Variable Naming Rules

Allowed

- ✓ `x`
- ✓ `name2`
- ✓ `street_address`
- ✓ `MyAge`
- ✓ `_rocket`

Not allowed

- ✗ `2d_map`
- ✗ `best-score`
- ✗ `school/work`
- ✗ `$Cost`
- ✗ `my variable`

Reserved Words

- There are reserved words which have special meanings in the Python language.
- Reserved words **can't be used** as variable names.

and	as	assert	break
class	continu e	def	del
elif	else	except	finally
False	for	from	global
if	import	in	is
lambda	nonloca l	None	not
or	pass	raise	return
True	try	with	while
yield			

Selecting Variable Names

- Variable names are for **programmers**, **not** computers.
  - Regardless of whether you name a variable `x`, `age`, or `zz23v`, Python will interpret your program in the same way.
  - However, good variable names can help make code more **understandable** to humans.

Selecting Variable Names

```python
s2sfg = 5
s2sgf = 10
s2sff = s2sfg * s2sgf
print(s2sff)
```

```python
width = 5
height = 10
area = width * height
print(area)
```

- The two programs above are equivalent to Python.
- But the one on the right is much easier to understand as a human!

Check Your Understanding

Q. Which of the following are valid variable names?

- `paper,scissors,rock`
- `_friend`
- `2_friends`
- `KEKWait`
- `if`

Check Your Understanding

Q. Which of the following are valid variable names?

- `paper,scissors,rock`
- `_friend`
- `2_friends`
- `KEKWait`
- `if`

A. `_friend` and `KEKWait`.

- Variable names can't contain **commas**.
- Variable names can't begin with a **digit**.
- if is a **reserved** word.

Ask the Audience

- What name might you give a variable which represents the cost of a product?

Statements

What is a Statement?

- A statement is a complete "step".
- A program consists of **multiple** statements.
- When you run a Python program, the statements within are executed in a well-defined **order**.
- Often a statement takes the form of a **line** of code.

What is a Statement?

- We have already seen a few different kinds of statements, for example:
  - `width = 5` is an assignment statement.
  - `print(area)` is a print statement.

Expression Statements

- Common uses:
  - Interacting with the Python interpreter.
  - Calling functions.
- More about expressions and functions in future lectures.

```
23
```

```
y - 5
```

```
max(100, 200)
```

Print Statements

- Technically a kind of expression statement.
- Used to **display** output text to the user.

```
print('Hello my friend!')
```

```
print(x * 2)
```

Assignment Statements

- Common uses:
    - **Defining** a variable.
    - **Updating** the contents of a variable.

- Assignment statements involve an **equals** sign.
    - Remember, this is **different** to = in mathematics!

```
minimum_age = 18
```

```
area = 2 * pi * radius
```

Input Statements.

- Technically a kind of **assignment** statement.
- **Used to accept input from the user**.
- Waits for the user to type input and **hit "Enter".**
- Input is **stored** in a variable.
- Optionally specify a **prompt** to show the user.

```
name = input()
```

```
c = input('Enter a colour: ')
```

Python Interactive Sessions vs. Scripts

Interactive interpreter session

- Start an interactive session with the `python` command.
- Write and execute a single statement at a time.
- If a statement produces a result, the result will be shown.

Python script

- Write multiple statements in a ".py" text file.
- Execute all of the statements together, for example:
  - `python my_script.py`
- If a statement produces a result, the result will **not** be shown.
  - **Use print statements to display results**.

Example: Interactive Session

```
>>> x = 5
>>> x + 2
7
>>> name = input('Enter your name: ')
Enter your name: Billy
>>> print('Hello ' + name)
Hello Billy
```

- Writing and running code is interleaved.
- Expression statements display results.

Example: Script

```
x = 5
x + 2
name = input('Enter your name: ')
print('Hello ' + name)
```

```
$ python my_script.py
Enter your name: Billy
Hello Billy
```

- First write all the code, then run all of the code.
- Expression statements so not display results unless printed.

Check Your Understanding

Q. What output(s) will be displayed after executing these statements in a Python script?

```python
age = 36
print(age)
age + 1
print('Done')
```

Check Your Understanding

Q. What output(s) will be displayed after executing these statements in a Python script?

A. 36 and Done.

- Only the print statements produce output messages.
- Even though line 3 contains a statement which produces a result, it is not shown.

```python
age = 36
print(age)
age=age + 1
print('Done')
```

Naming Conventions and Comments

Meaningful Variable Names

- It is good practice to give variables meaningful names.
- Meaningful names are for the benefit of human programmers only.
  - Naming a variable "interest_rate" won't magically give Python the ability to calculate interest.
- Spending a few moments to select good variable names can greatly increase code readability.
  - **This saves time in the long run!**

Example: Meaningful Variable Names

```
variable1 = 10
variable2 = 5
variable3 = 20
variable4 = variable1 * variable2
if variable4 > variable3:
    print('Danger!')
```

```
acceleration = 10
acceleration_time = 5
maximum_safe_speed = 20
speed = acceleration * acceleration_time
if speed > maximum_safe_speed:
    print('Danger!')
```

- The names of the variables in the second program are **meaningful** to humans.
- When the meaning of a program is unclear, as in the first program, the programmer is more likely to make **mistakes**.

Common Python Naming Conventions

- Multiple words in a variable name are separated with_underscores.

- Ordinary variables are named in **lowercase**, with **underscores separating** words.
  - e.g. `current_score = 0`

- **Constants** (variables with values that are never updated) are named in **UPPERCASE**.
  - e.g. `EARTH_GRAVITY = 9.81`

Other Naming Conventions

- Where appropriate, you may want to adopt your own naming conventions.
  - Indicate units:
    - `time` → `time_secs`
  - Indicate that a count is being stored:
    - `participants` → `num_participants`
  - Indicate subtle differences:
    - `password1`, `password2` → `old_password`, `new_password`

Comments

- Comments are sections of code ignored by Python.
- Can be used to annotate code in plain English (good variable names aren't always enough)
- In Python, a comment starts with a **hash** (**#**) and continues to the end of the line.

```
# This is a comment on its own line.

x = 5 # This comment shares a line with actual code.

# You can't place a comment before code # y = 7
```

Common Uses for Comments

- **Explain** to other **programmers** (or your future self) how a complex section of code works.
- Leave **notes** about **future** work to do.
- Add a **copyright** notice or indicate authorship.
- Temporarily **disable** a section of code ("commenting out" code).

Comments in the Wild: Description

```python
def try_destroy_gl_object(self, obj):
    if self.window is None:
        # Can't destroy the object if its context has already been destroyed.
        return False
    # If an object gets garbage collected while another context is current, temporarily make
    # this context current and destroy the object.
    with self:
        obj.destroy()
    return True
```

Don't be afraid of writing comments which are longer than the associated code if you deem it appropriate!

Comments in the Wild: Future Work

```python
        ax3 = fig.add_subplot(4, 1, 3)
        ax4 = fig.add_subplot(4, 1, 4)

        # TODO: Add histogram for example index
        stats = {
            'root_x': StatTracker(np.linspace(-1.0, 1.0, 100)),
            'root_y': StatTracker(np.linspace(-1.0, 1.0, 100)),
```

Future work comments often start with "TODO:" (as in "this is future work **to do**").

Comments in the Wild: License Information

```python
# Copyright 2017 Aiden Nibali
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
DSNT (soft-argmax) operations for use in PyTorch computation graphs.
"""

from functools import reduce
from operator import mul

import torch
```

"Commenting Out" Code

- Since comments are effectively **ignored** by Python, they can be used to **disable** lines of code.
  - Simply add a "**#**" to the start of the line.
- The program below outputs 5 (not 6) since line **2** is "commented out".

```python
x = 5
# x = x + 1
print(x)
```

Check Your Understanding

Q. What result will be displayed by the program?

```
x = 2
five = x + 2
y = five + 4
# y = y + 1
print(y)
```

Check Your Understanding

Q. What result will be displayed by the program?

A. 8.

- Naming a variable "five" does not affect its value.
- The commented line is not executed (line 4 is ignored).

```python
x = 2
five = x + 2
y = five + 4
# y = y + 1
print(y)
```

4

Expressions

What Are Expressions?

- An expression is a small piece of code that **resolves to a value**.

- Some extremely simple examples are:
  - `42` (a literal resolves to **itself**).
  - `my_var` (a variable **resolves** to its **value**).

What Are Expressions?

- The **highlighted** parts of the code shown on the right are expressions.

- Note that only the **right** hand side of assignment is an **expression**.
  - The **left** hand side must be an **identifier** (e.g. a variable name).

```
>>> 33.33
33.33
>>> artist = 'Avenade'
>>> albums = 1
>>> albums
1
>>> new_albums = albums + 1
>>> new_albums
2
```

What Are Expressions?

- Expressions can use **operators** to compute a result.
- A combination of numeric values, operators, and sometimes parenthesis **()**.
  - `3.5 - 0.75` (resolves to 2.75).
  - `x + y` (resolves to the **sum** of **x** and **y**).
  - `r + 1` (resolves to the value of **r** plus one).

Operators

Numeric Operators

OPERATOR	NAME
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`**`	Exponentiation
`%`	Modulo
`//`	Integer division

- An asterisk (*) denotes multiplication.
  - `2 * 3` (gives 6)
- A forward slash (/) denotes division.
  - `5 / 2` (gives 2.5)
- `//` Integer division
  - `5 / 2` (gives **2**)
- A double asterisk (**) denotes "to the power of".
  - `2 ** 3` (gives 8)

Integer Division and Modulo

- Recall performing division by hand in school.
  - We would get two results: the quotient, and the remainder.

- For example, when evaluating 7 ÷ 2, we might say: "2 goes into 7 three times, with a remainder of one".
  - The quotient is 3.
  - The remainder is 1.

Integer Division and Modulo

- In Python:
  - Integer division (//) calculates the quotient.
  - Modulo (%) calculates the remainder.

- Examples:
  - `7 // 2` (gives 3)
  - `7 % 2` (gives 1)

- Can also think of integer division as regular division with the result rounded down.

Combining Expressions

- Multiple expressions can be combined:
  - e.g. `3 + 2 * 6 - 1`
- **But** how does Python **decide** which **operator goes first**?
  - In the above example, how does Python choose whether to do `3 + 2`, `2 * 6`, or `6 - 1` first?

Operator Precedence

- Each of the operators in Python has a certain precedence ("**priority**").
- This is similar to mathematics.
  - Remember learning PEMDAS (**Parentheses**, **Exponents**, **Multiplication**/**Division**, **Addition**/**Subtraction**) in school?
- Operators with higher precedence are evaluated first.
- Operators with the same precedence are evaluated **left-to-right**.

Operator Precedence

OPERATOR		NAME
( )	Higher	Parentheses
**		Exponentiation
*, /, %, //		Multiplication, etc.
+, -	Lower	Addition, etc.

- Similar order to mathematics (PEMDAS).
- Since **parentheses** have highest precedence, you can always explicitly group things.

Check Your Understanding

Q. What does the expression `2 + 5 * 7 // (1 + 1)` evaluate to?

OPERATOR	NAME
`()`	Parentheses
`**`	Exponentiation
`*`, `/`, `%`, `//`	Multiplication, etc.
`+`, `-`	Addition, etc.

Check Your Understanding

Q. What does the expression `2 + 5 * 7 // (1 + 1)` evaluate to?

A. 19.

1. 2 + 5 * 7 // (1 + 1)
2. 2 + 5 * 7 // 2
3. 2 + 35 // 2
4. 2 + 17
5. 19

OPERATOR	NAME
`()`	Parentheses
`**`	Exponentiation
`*`, `/`, `%`, `//`	Multiplication, etc.
`+`, `-`	Addition, etc.

Types

Primitive Types

In Python, we can assign different **values** of different **variables**, and different types can do different things. The assigned values are known as Data Types.

TYPE	NAME	DESCRIPTION	EXAMPLE
`str`	String	A string of characters (text).	`'Hello'`
`int`	Integer	A whole number.	`-43`
`float`	Float	A number with a decimal point.	`23.6`
`bool`	Boolean	A boolean (true/false) value.	`True`

Today we will discuss **integers**, **floats**, and **strings**. We will cover booleans in the **next** lecture.

Different Types of Numbers

- Integers (type int) represent whole numbers.
- Often used to represent **counts** of things:
  - Inventory **levels**
  - **Age** (in years)
  - **Attendance**
- -43, 0, and 10000 are all **integers**.

- Floating point numbers, or "floats", (type float) have a decimal point and fractional part.
- Often used to represent precise measurements:
  - **Temperature**
  - **Percentages**
  - **Prices**.
- **22.5**, 0.0, and -0.9999 are all floats.

Strings

- **String** (type str) literals (snippets of text) are written between matched **quotation marks**.
    - e.g. `'text'` or `"text"`.

```
>>> x = "1"
>>> type(x)
<class 'str'>
>>> type('hi')
<class 'str'>
>>> x=20
>>> print (str(x))
>>>'20'
```

String Concatenation

- String concatenation means joining two pieces of text ("strings") together.
  - `'bed' + 'room'` (gives 'bedroom')

- Can be used to build a message to show the user.

```
>>> name = 'Jeremiah'
>>> print('Hello ' + name)
Hello Jeremiah
```

String Concatenation

Beware!

You can't combine a string and a number using the "**+**" operator. We'll discuss how to avoid this error in the next part of the lecture as we learn about *types*.

```
>>> 'My age is: ' + 28
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Determining Types

- Python can tell us the type of a value.
- Simply use the `type()` function.
- Especially useful in interactive sessions.

```
>>> x = 1
>>> type(x)
<class 'int'>
>>> type(3.4)
<class 'float'>
>>> type(x + 3.4)
<class 'float’>
>>> type(“Hi CSE4P”)
<class ‘str’>
```

Different Types Behave Differently

- There is a difference between `20`, `20.0`, and `'20'`:
  - `20` is an **integer**.
  - `20.0` is a **float**.
  - `'20'` is a **string**.
- Python sees these values ***differently***.
- Values of different types **behave** *differently*.
  - Python chooses what operators do based on **types**.

Check Your Understanding

Q. What does the expression `'1' + '2'` evaluate to?

Check Your Understanding

Q. What does the expression `'1' + '2'` evaluate to?

A. The string `'12'`.

Since both '1' and '2' are strings---not floats or integers---Python treats them as text and not numbers. So, when Python sees the + operator, it performs string concatenation instead of numerical addition. This is the same behaviour as our earlier example of 'bed' + 'room'.

Automatic Type Conversion

- Type conversion is the process of deriving a value of a different type from an existing value.
- Sometimes Python will **automatically** convert types.
- For example, when an **integer** and a **float** are **added**, the integer is automatically **converted** into a **float**.

```
>>> x = 1
>>> x + 3.4  # Python converts the 1 to 1.0
4.4
```

Explicit Type Conversion

- Python doesn't always know how types should be converted, so there are times when you need to convert types explicitly.

- This can be achieved using the **int()**, **float()**, **str()**, and **bool()** functions.

Converting Numbers to Strings

- Converting numbers to strings can be useful for building output messages.

```
>>> age = 28
>>> 'My age is: ' + str(age)
'My age is: 28'
```

- Converting a float to a string will always include the decimal part, even if it is zero.

```
>>> str(420.0)
'420.0'
```

Converting Strings to Numbers

- Converting strings to numbers can be useful for accepting input.

```
>>> age = input('Enter your age: ')
Enter your age: 28
>>> int(age) + 7

35
```

- If the string does not represent a number, Python will raise an error.

```
>>> float('apple')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'apple'
```

Check Your Understanding

Q. What value does the following expression evaluate to?

```
str(9 + 1.0) + '0'
```

Check Your Understanding

Q. What value does the following expression evaluate to?

```
str(9 + 1.0) + '0'
```

A. The string `'10.00'`.

1. Python implicitly converts the 9 to 9.0, and adds 1.0 to get 10.0.

2. The str() function converts the float 10.0 to the string '10.0’.

3. Finally, the string '10.0' is concatenated with '0' to get '10.00'.

Example Program: Temperature Converter

Task Definition

Create a program which converts a temperature value from degrees Celsius (C) to degrees Fahrenheit (F) using the following conversion formula, where C is the temperature in degrees Celsius and F is the temperature in degrees

Fahrenheit:

$$F = \frac{9C}{5} + 32$$

Identifying Inputs and Outputs

- **Input**: C, the temperature in degrees Celsius.
- **Output**: F, the temperature in degrees Fahrenheit.
- Examples:
  - 0 °C → 32 °F
  - 100 °C → 212 °F
  - 22.5 °C → 68.9 °F

$$F = \frac{9C}{5} + 32$$

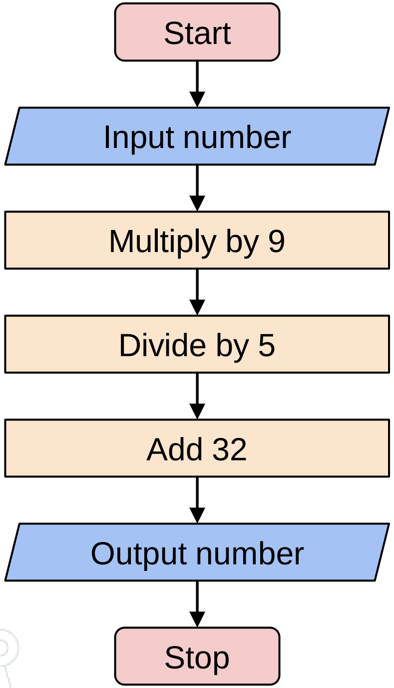
Identifying Processing Steps

Example

- Input: 100 → C=100
- Processing:
  1. $100 \times 9 = 900$
  2. $900 \div 5 = 180$
  3. $180 + 32 = 212$
- Output: 212

$$F = \frac{9C}{5} + 32$$

Drawing a Flowchart

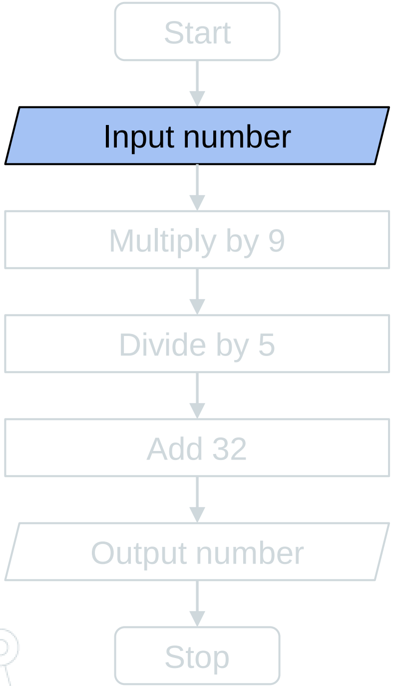


$$F = \frac{9C}{5} + 32$$

Example

- Input: 100
- Processing:
  1. $100 \times 9 = 900$
  2. $900 \div 5 = 180$
  3. $180 + 32 = 212$
- Output: 212

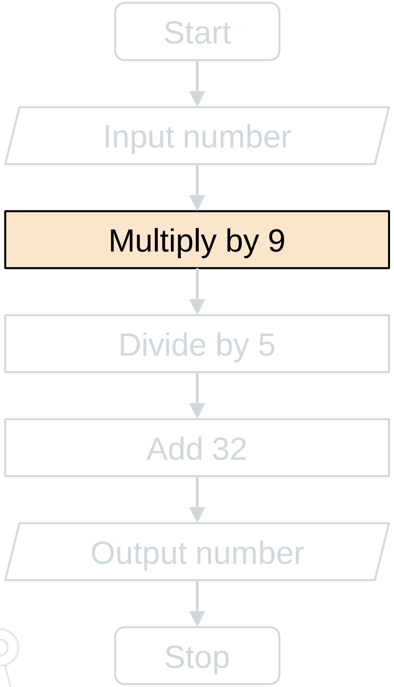
Translating to Python Code




```
tc = input('Enter Celsius: ')
x = float(tc)
```

- The **input** is initially **read** in as a **string** (type str).
- We **convert** it into a number (type float).

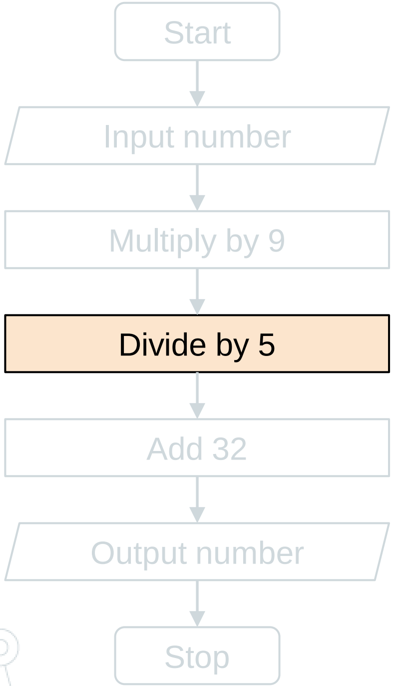
Translating to Python Code




```
x = x * 9
```

- The asterisk (*) means "multiply" in Python.

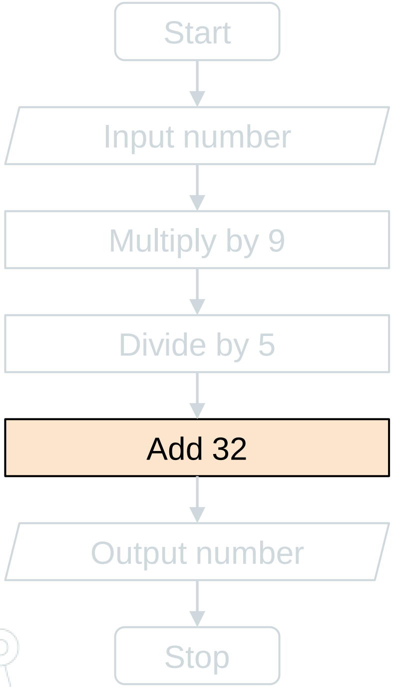
Translating to Python Code




```
x = x / 5
```

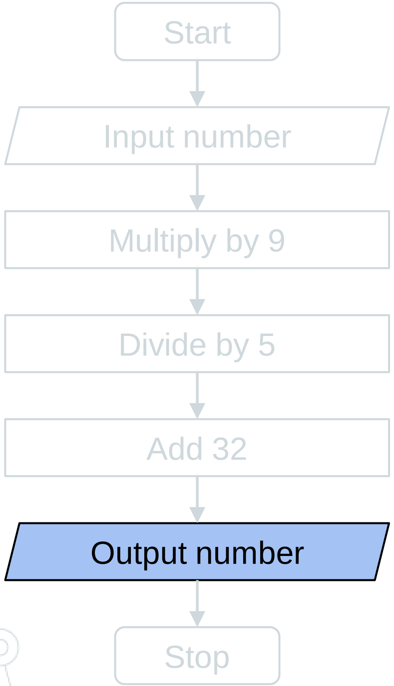
- The forward slash (/) means "divide" in Python.

Translating to Python Code




```python
x = x + 32
```

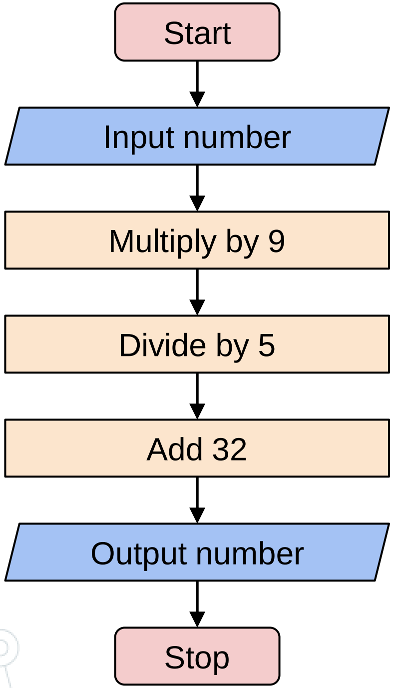
Translating to Python Code




```python
tf = str(x)
print('Fahrenheit: ' + tf)
```

- We convert our number (type float) into a string (type str) so that we can build our output message.

The Final Program




```python
# Input
tc = input('Enter Celsius: ')
x = float(tc)
# Processing
x = x * 9   # Multiply by 9
x = x / 5   # Divide by 5
x = x + 32  # Add 32
# Output
tf = str(x)
print('Fahrenheit: ' + tf)
```

Check Your Understanding

Q. After reading user input, we converted the value into a float:

```
x = float(tc)
```

Why didn't we convert to an integer instead?

```
x = int(tc)
```

Check Your Understanding

Q. After reading user input, we converted the value into a float:

```
x = float(tc)
```

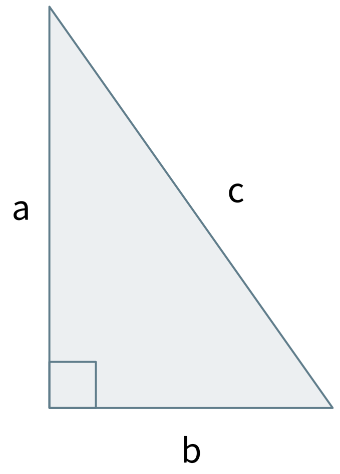
Why didn't we convert to an integer instead?

```
x = int(tc)
```

A. Type int only supports whole numbers. Therefore using int() would not allow the user to input fractional temperatures, like 20.5.

Live Coding Demo

- Let's say that you have to calculate a bunch of hypotenuse lengths (a totally normal thing that normal people have to do).
- You could use a calculator like a peasant, or you could write a program like a non-gender-specific member of the royal family! 😅



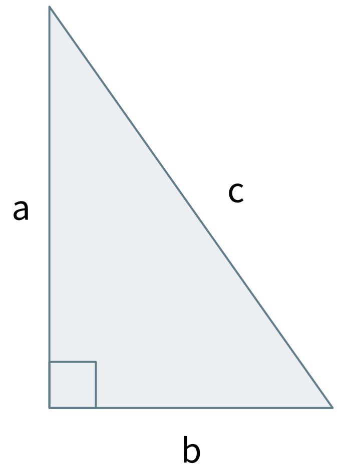
$$a^2 + b^2 = c^2$$

$$\Rightarrow c = \sqrt{a^2 + b^2}$$

Live Coding Demo

```python
# File: hypotenuse.py
a = float(input('Enter a: '))
b = float(input('Enter b: '))
c = (a ** 2 + b ** 2) ** 0.5
print('Hypotenuse: ' + str(c))
```

```
$ python hypotenuse.py
Enter a: 3
Enter b: 4
Hypotenuse: 5.0
```



$$a^2 + b^2 = c^2$$

$$\Rightarrow c = \sqrt{a^2 + b^2}$$

Thanks for your attention!

The lecture recording will be made available on LMS.

PPT Acknowledgement: Dr Aiden Nibali, CS&IT LTU.