Lecture 2.1

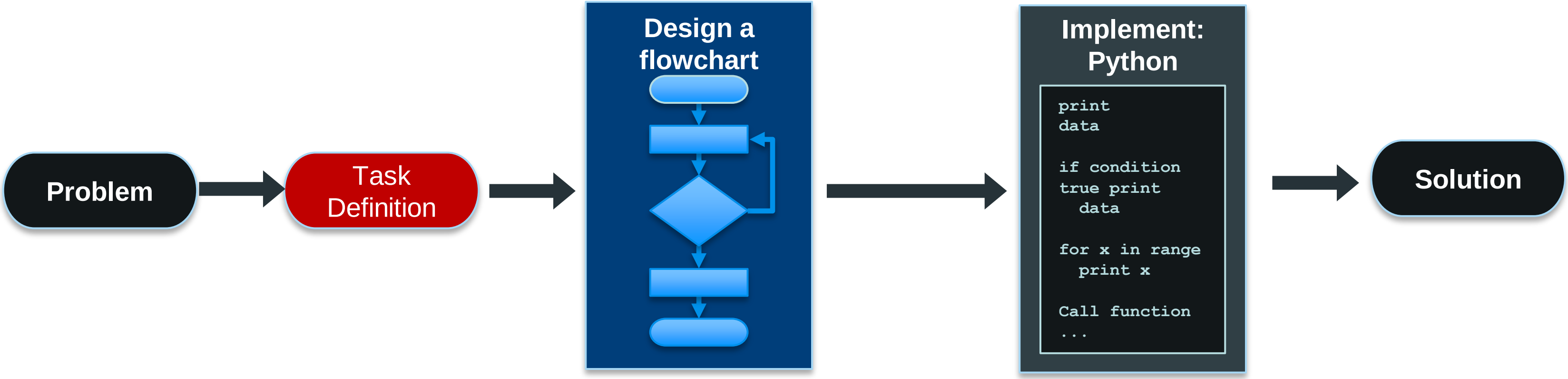
Comparisons and Boolean Logic

We have covered in previous weeks (Week 1)

- Flowcharts
- How to install and run Python editor
- Program Development steps
- **Input**, **Processing**, and **Output**
- **Variables**, Statements, and Comments
- Expressions and Data Types

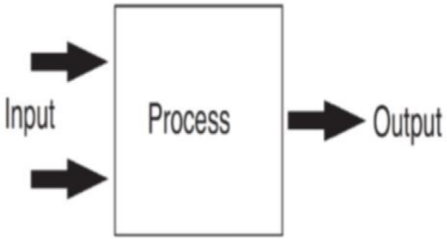
Program Development steps

- 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)



```
print
data

if condition
true print
  data

for x in range
  print x

Call function
...
```

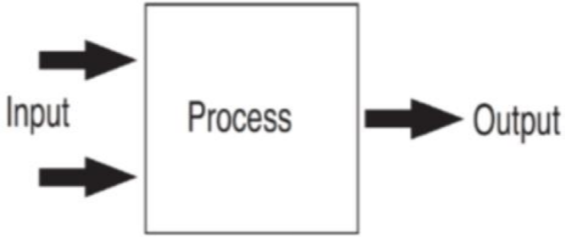
Python Input



- Take Input from User – direct or keyboard
- Take input from File – text, csv, binary, etc
- Take input from System – click, button, drop-down, mouse

Python Input

◎ Take Input from User – Direct



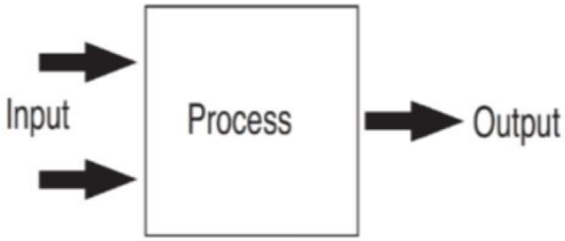
```
>>> x = 10      # integer
>>> y = 0.6     # float
>>> z = “Hello” # string
```

```
>>> x = 10      # integer
>>> y = 0.6     # float
>>> z = x + y
```

```
>>> x = “Hello”      # string
>>> y = “CSE4IP”     # string
>>> z = x + y
```

Python Input

◎ Take Input from User – keyboard



Python uses **input()** function to get input from the user - **Keyboard**

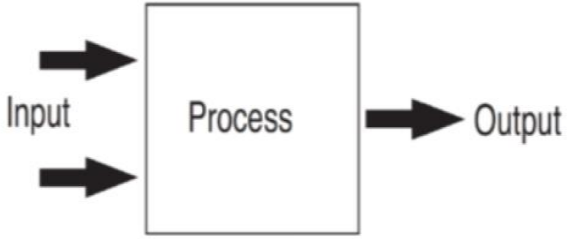
Syntax:

- **input(“*prompt*”)**

- ***prompt*** : a string - a default message displayed before the input

Python Input

◎ Take Input from User – keyboard



input(“*prompt*”)

Example: take three different inputs from user.

```
>>>  x  =  input (“Enter a number:  ”)
  Enter a number: 5

>>>  y  =  input (“Enter a number:  ”)
  Enter a number: 8.3

>>>  z  = input (“Enter a string:  ”)
  Enter a string: Hello CSE4IP
```

Python Input

Take Input from User – keyboard: input(*prompt*)

Example: take three different inputs from user.

```
>>>  x  =  input (“Enter a number:  ”)
  Enter a number: 5
>>> X
‘5’

>>>  y  =  input (“Enter a number:  ”)
  Enter a number: 8.3
>>> y
‘8.3’

>>>  z  = input (“Enter a string:  ”)
   Enter a string: Hello CSE4IP
>>> Z
‘Hello CSE4IP’
```

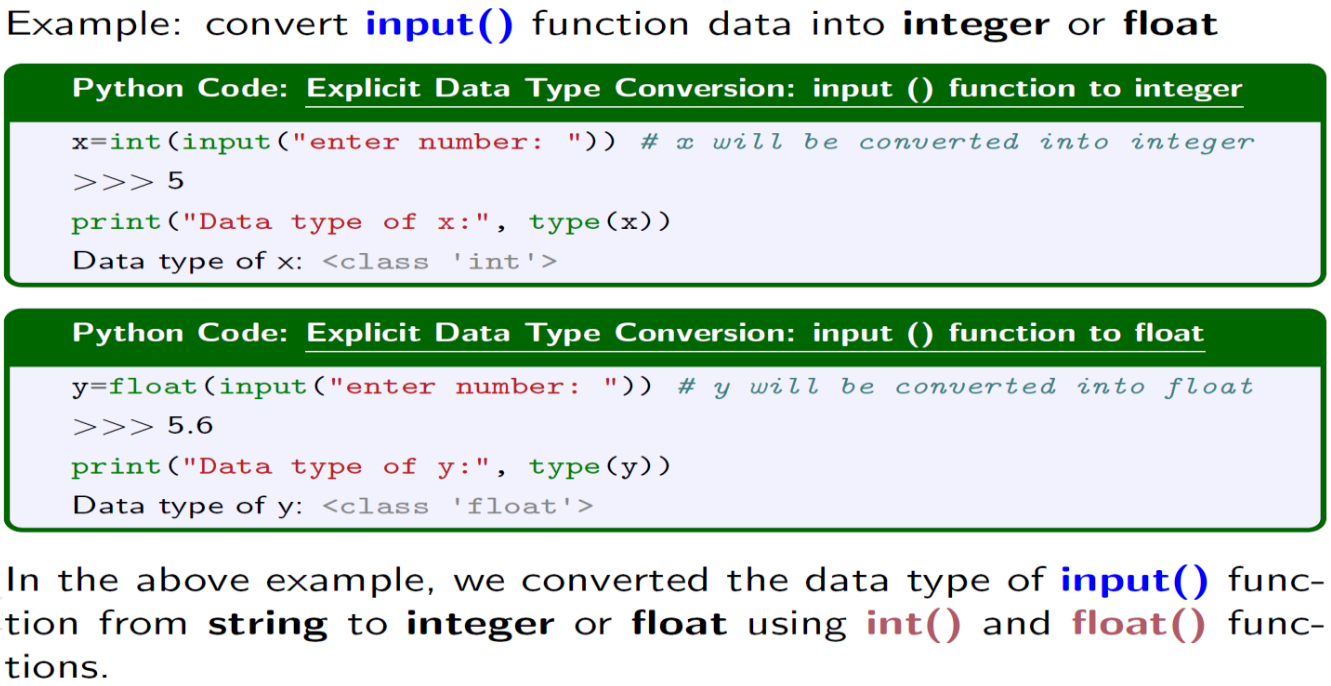
- We can see that all inputs are saved as string (‘’)

- We can use Explicit Type Conversion functions (**covered in Lecture 2.2**) to convert them into a proper data type.

- `int()`
- `float()`
- `str()`

Python Input

Take Input from User – keyboard: input(*prompt*)



Example: convert **input()** function data into **integer** or **float**

Python Code: Explicit Data Type Conversion: input () function to integer

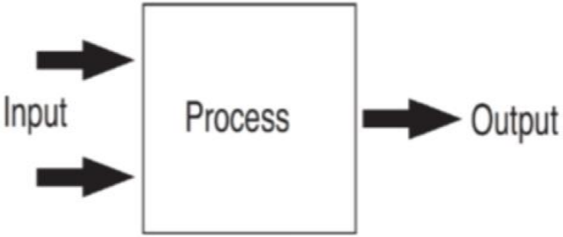
```
x=int(input("enter number: ")) # x will be converted into integer
>>> 5
print("Data type of x:", type(x))
Data type of x: <class 'int'>
```

Python Code: Explicit Data Type Conversion: input () function to float

```
y=float(input("enter number: ")) # y will be converted into float
>>> 5.6
print("Data type of y:", type(y))
Data type of y: <class 'float'>
```

In the above example, we converted the data type of **input()** function from **string** to **integer** or **float** using **int()** and **float()** functions.

Python Output



- **Display the output of a program to the standard output device - screen and console**

- **Save the output in a file – text, csv, binary, …, etc.**

Python Output

Python uses `print()` function to display the output of a program into screen.

Python Code: print () function

```
>>> print("Display CSE4IP at the screen")
Display CSE4IP at the screen

>>> a = 10
>>> print('The value of a is', a)
The value of a is 10

>>> a = 10
>>> b=3
>>> c=a-b
>>> print('The value of c is', c)
The value of c is 7

>>> a = 10
>>> b=4
>>> print('a-b=', a-b)
a-b= 6
>>> print('10 + 30=', 10+30)
10 + 30= 40
```

Comparisons and Boolean Logic

Topic 2.1 Intended Learning Outcomes

- By the end of week 2 you should be able to:
  - Write boolean expressions for questions with **yes/no** answers, and
  - Use selection control structures to specify different flows through a program.

Lecture Overview

1. Booleans
2. Comparison Operators
3. Logical Operators

Booleans

Review of Types

- We have encountered three data types so far:
  - str (string): A string of characters (i.e. text).
    - e.g. `‘CSE4IP'`, `"August"`.
  - int (integer): A whole number.
    - e.g. `23`, `-1000`
  - float (floating point number): A number with a fractional part.
    - e.g. `23.45`, `-5.0`

How About **Yes/No** Values?

- How would we store the answer to a yes/no question?
  - e.g. Is the video paused?

- Could use strings:
  - `is_paused = 'yes'`
  - Quickly gets confusing. Does "YES" also mean yes? "Y"? "True"?

- Could use integers (0 means no, 1 means yes).
  - `is_paused = 1`
  - Not very readable.

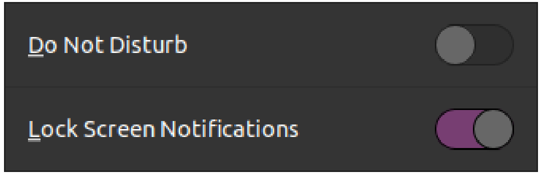
Introducing Booleans

- A boolean represents a binary value.
  - **Yes**/no, **true**/false, **on**/off.

- There are only **two** possible **boolean** values.

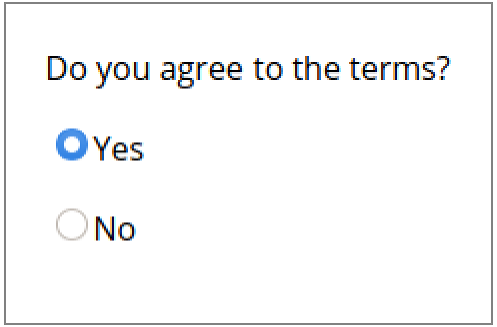
- Useful for representing answers to **yes**/**no** questions.
  - Did the user opt into the newsletter?
  - Is the car new?
  - Is the average test score greater than 50?

Visual Representations of Booleans



Do Not Disturb

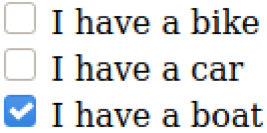
Lock Screen Notifications



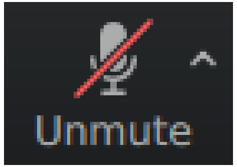
Do you agree to the terms?

● Yes

○ No



□ I have a bike
□ I have a car
☑ I have a boat



Booleans in Python

- In Python, booleans have the bool type.

- The two boolean literals in Python are `True` and `False`.
    - These are the *only* two values of type bool.

- For the previous example:
    - `is_paused = True`

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

Converting Values to Booleans

- Values of other types can be converted into booleans.
- A value that converts to **True** is said to be truthy.
- A value that converts to **False** is said to be falsy.

Truthy and Falsy: Numbers

- For numeric types:
  - Zero is falsy.
  - All other values are truthy.

```
>>> bool(0)
False
>>> bool(-42)
True
>>> bool(0.00)
False
>>> bool(0.0001)
True
```

Truthy and Falsy: Strings

- For strings:
  - The empty string is falsy.
  - All other values are truthy.
- The **empty** string is a **string** with **no characters** in it.

```
>>> bool('hello')
True
>>> bool('')
False
>>> bool('False')
True
>>> bool(' ')
True
>>> bool("")

False
```

Check Your Understanding

Q. Is the string `'No'` truthy or falsy?

Check Your Understanding

Q. Is the string `'No'` truthy or falsy?

A. Truthy.

The only falsy string is the empty string. Python does not attempt to read the contents of the string beyond checking if it is empty.

Comparison Operators

Boolean Expressions

- Recall that a numeric expression is an expression which evaluates to a number (e.g. `2 + 2`).

- A boolean expression is an expression which evaluates to a boolean (i.e. True or False).

- A simple kind of boolean expression is **comparing** two values. For example:
  - Is one number **greater** than another?
  - Are two strings **equal**?

Comparison Operators

- Two values can be **compared** using a comparison operator.

- The result of a comparison will be **True** or **False**, depending on whether the condition is **satisfied**.

- Note that Python uses double equals (`==`) to check for equality.

PYTHON	MATHS	CONDITION
`x == y`	x = y	x is equal to y
`x != y`	x ≠ y	x is not equal to y
`x > y`	x > y	x is greater than y
`x < y`	x < y	x is less than y
`x >= y`	x ≥ y	x is greater than or equal to y
`x <= y`	x ≤ y	x is less than or equal to y

Numeric Comparisons

- Numeric comparisons work as you would expect from mathematics.

```
>>> x = 5
>>> x > 2
True
>>> x > 5
False
>>> x >= 5
True
>>> y = 7.5
>>> x < y
True
>>> -50 > y
False
```

String Comparisons

- String comparisons are based on lexicographical ordering.
  - This is similar to ***dictionary ordering***.
- However, all **uppercase letters come** before **lowercase** letters.

```
>>> word = 'apple'

>>> word > 'banana'
False

>>> word == 'apple'
True

>>> word < 'apple pie'
True

>>> word < 'Zucchini'

False
```

Comparisons Across Types

Beware!

Avoid comparing values of different types (e.g. a number and a string)---the results are probably not what you expect.

```
>>> 5 == '5'
False

>>> 0 < '5'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

Check Your Understanding

Q. What is the Python boolean expression for "twice the value of x is not equal to 100"?

Check Your Understanding

Q. What is the Python boolean expression for "twice the value of **x** is **not** equal to 100"?

A. `x * 2 != 100`

Or, equivalently:

- `2 * x != 100`
- `100 != x * 2`
- `100 != 2 * x`

Logical Operators

Logical Operators

- Logical operators can be used to **combine** and **modify** **boolean** expressions.

- Python has **three** logical **operators**: `and`, `or`, `not`.
  - These generally behave as you would expect from the meaning of the words.

The and Operator (Logical And)

- True if both expressions are truthy.
- False if at least one expression is falsy.

P	Q	P and Q
True	True	True
True	False	False
False	True	False
False	False	False

```
>>> x = 5
>>> y = 7.5

>>> x < y and x == 5
True

>>> y == 5 and x == 5
False

>>> x == 0 and y <= 0
False
```

The `or` Operator (Logical Or)

- True if at least one expression is truthy.
- False if both expressions are falsy.

P	Q	P or Q
True	True	True
True	False	True
False	True	True
False	False	False

```
>>> x = 5
>>> y = 7.5

>>> x < y or x == 5
True

>>> y == 5 or x == 5
True

>>> x == 0 or y <= 0
False
```

The not Operator (Logical Not)

- Negates the expression.
- True if the expression is falsy.
- False if the expression is truthy.

P	not P
True	False
False	True

```
>>> x = 5
>>> y = 7.5

>>> not x < y
False

>>> not y == 5
True
```

Combining Boolean Expressions

- **Logical** operators can be used to build **complex** boolean expressions.

- Other kinds of **operators** can be included in the expression.

- For example, "**either** **x** is **greater** than **10** and **even**, **or** **y** is **less** than **0**" can be expressed as:

```
(x > 10 and x % 2 == 0) or y < 0
```

Extended Operator Precedence

OPERATOR		NAME
`()`	Higher	Parentheses
`**`		Exponentiation
`*`, `/`, `%`, `//`		Multiplication, etc.
`+`, `-`		Addition, etc.
`==`, `!=`, `>`, `<`, `>=`, `<=`		Comparison
`not`		Logical "not"
`and`		Logical "and"
`or`	Lower	Logical "or"

Check Your Understanding

Q. What does the following expression evaluate to?

```
2 * 7 > 5 and not 27 < 8
```

OPERATOR		NAME
`()`	Higher	Parentheses
`**`		Exponentiation
`*`, `/`, `%`, `//`		Multiplication, etc.
`+`, `-`		Addition, etc.
`==`, `!=`, `>`, `<`, `>=`, `<=`		Comparison
`not`		Logical "not"
`and`		Logical "and"
`or`	Lower	Logical "or"

Check Your Understanding

Q. What does the following expression evaluate to?

```
2 * 7 > 5 and not 27 < 8
```

A. True.

1. 2 * 7 > 5 and not 27 < 8
2. 14 > 5 and not 27 < 8
3. True and not 27 < 8
4. True and not False
5. True and True
6. True

OPERATOR		NAME
`( )`	Higher	Parentheses
`**`		Exponentiation
`*`, `/`, `%`, `//`		Multiplication, etc.
`+`, `-`		Addition, etc.
`==`, `!=`, `>`, `<`, `>=`, `<=`		Comparison
`not`		Logical "not"
`and`		Logical "and"
`or`	Lower	Logical "or"

Lecture 2.2

Conditional Execution

Lecture Overview

1. Control Flow
2. Selection Statements
3. Conditional Execution Pitfalls

Control Flow

Control Flow

- Control flow describes the **order** in which the **statements** making up a program are **executed**.

- So far all of the Python code we've looked at has had ***sequential*** control flow.

- However, there are a few different control structures which can result in **different** kinds of control flow.

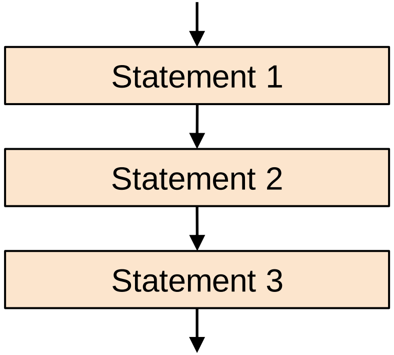
Control Flow: **Sequence (or sequential)**

- The most basic control structure is a **sequence** of statements.

- Statements in a sequence are executed in order of **appearance**.

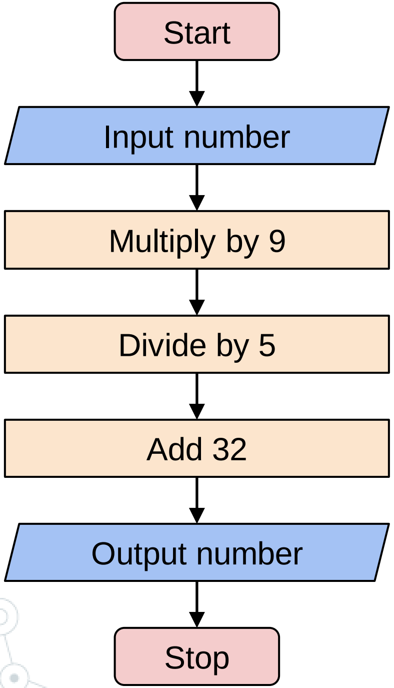
- There is only **one** possible **path** for control flow.

- Represented by a series of process/input/output elements in a flowchart.



Control Flow: **Sequence (or sequential)**

Example Program: Temperature Converter



```
# 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)
```

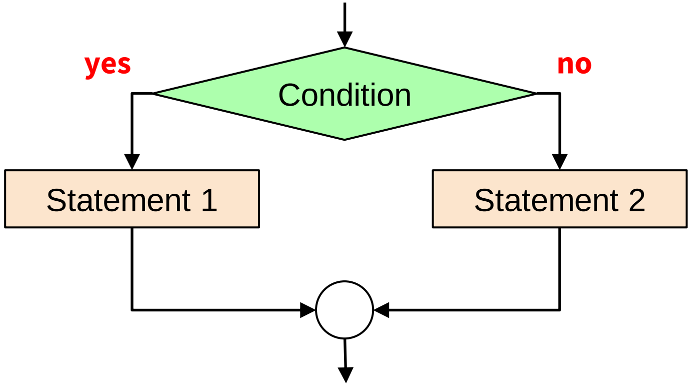
Control Flow: **Selection**

- In a selection control structure, a **condition** determines which statements are **executed**.

- A selection introduces **multiple paths** for control flow to take, from which **one** will be **selected**.

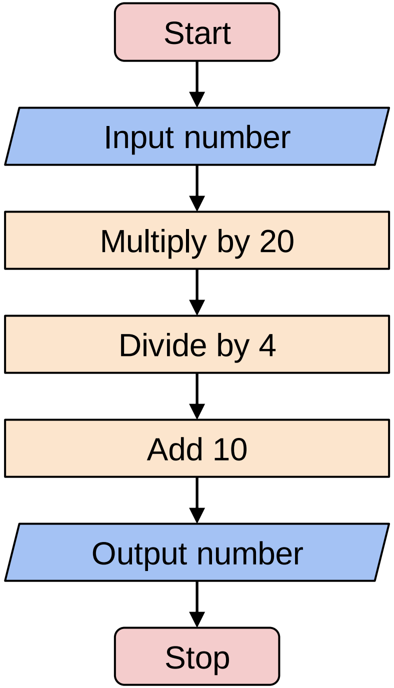
- This allows the program to make a **decision**.

- Represented by a decision element in a flowchart.



Check Your Understanding

Q. Does the flowchart show a **sequence** or **selection** control structure?

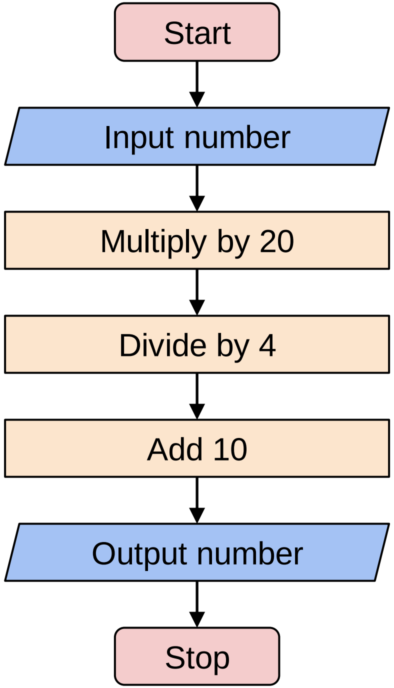


Check Your Understanding

Q. Does the flowchart show a **sequence** or **selection** control structure?

A. Sequence.

There is only one possible path for control flow to take.



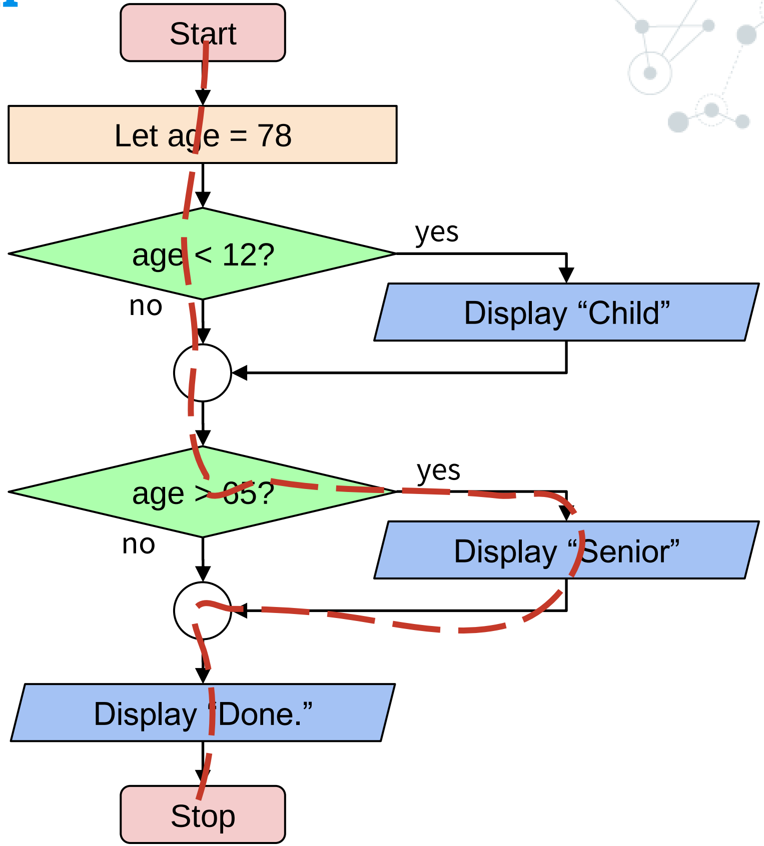
Visualising Control Flow

- Drawing the control flow over a flowchart can be helpful when **reasoning** about a program's **logic**.

- When drawn in this way, control flow is a single **path** from the **start** element to the **stop** element.
    - **The control flow never splits**.
    - When a **decision** element is **encountered**, **one branch** is **selected**.

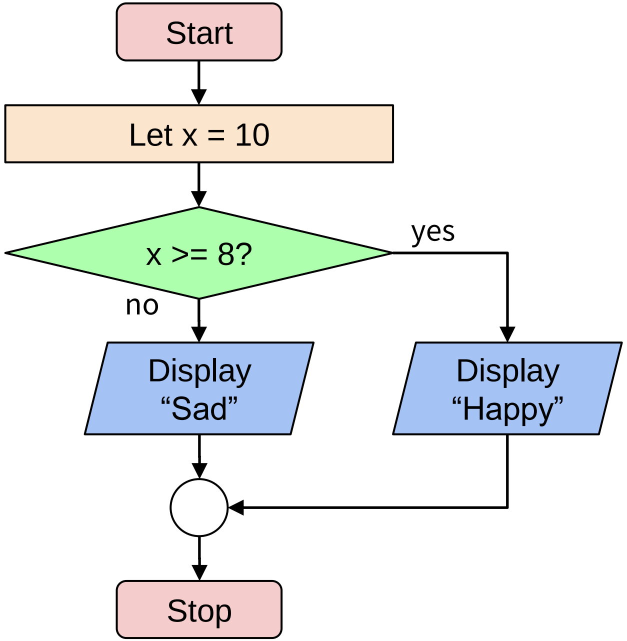
Control Flow Example: **Selection**

1. Begin at the start element.
2. The first condition determines which path the control flow will take.
3. The condition is not met, so the “**no**” path is taken.
4. The second condition is met, so the “**yes**” path is taken.
5. The control flow continues to the stop element.



Check Your Understanding

Q. What is the correct control flow for this flowchart?

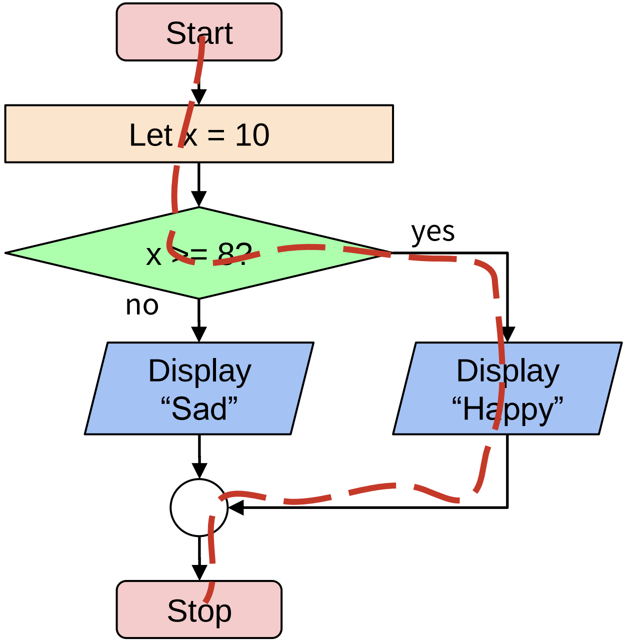


Check Your Understanding

Q. What is the correct control flow for this flowchart?

A. See image.

- Control flow can't split.
- The condition x >= 8 is met, so the "yes" branch is taken.



Selection Statements

Selection Statements

- Selection statements are used to define **selection** control structures.

- When coding in Python, **indentation** **MUST** be used to ***group*** **statements** under a particular **selection** statement.

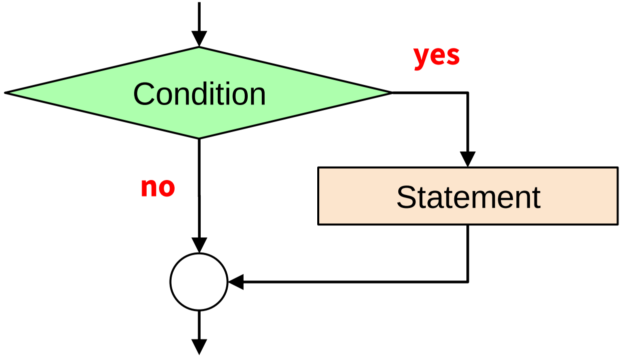
Selection Statements

- Python has **three** main **selection** statements:
  - `if` statements, which begin a selection control structure by defining a sequence of statements to execute if a condition is met.
  - `elif` and `else` statements, which optionally define **alternative** code **paths** (or "branches") in the selection control structure.

Simple Conditional Execution (`if`)

```
if Condition:
    Statement
```

- If the condition **is truthy**:
  - Execute the statement(s) in the if block.
- If the condition **is falsy**:
  - Do nothing.



Example: Simple Conditional Execution

Task definition

Write a program which subtracts 40% tax from a person's income when their pre-tax income is above $100,000. Display the result.

- **Input**: pre-tax income.
- **Output**: post-tax income.
- **If** pre-tax income is **greater** than $100,000, then the **processing** steps are:
  - **Calculate** tax as 40% of the pre-tax income.
  - **Subtract** tax from the income.

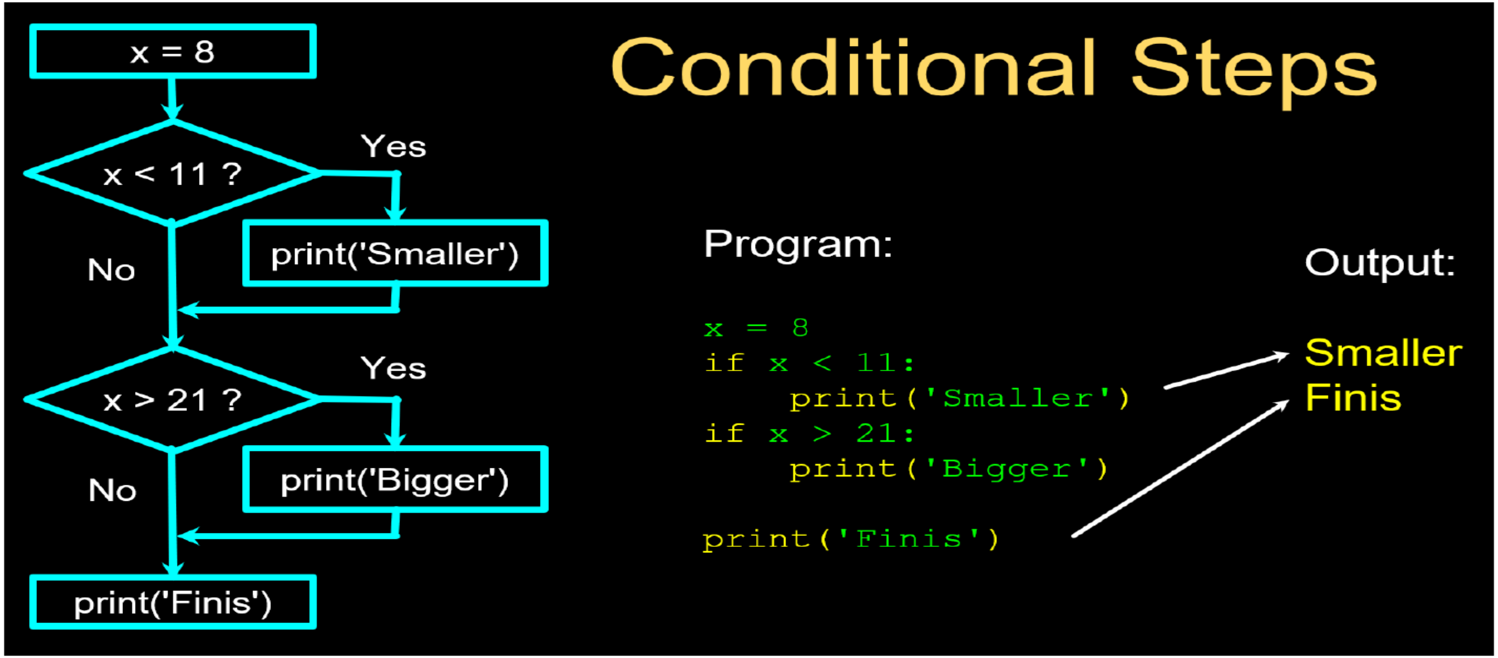
Example: Simple Conditional Execution

```
# File: tax1.py
income = float(input('Enter income: $'))
if income > 100000:
    tax = 0.4 * income
    income = income - tax
print(income)
```

- **Remember**: indentation is important!
- If the condition evaluates to:
  - **True**, then the indented statements are executed.
  - **False**, then the indented statements are skipped.

Conditional Execution - **Multiple if statements**

We can use **multiple if statements** to check several conditions and take the actions based on the stratified ones.



Conditional Steps

Program:

```
x = 8
if x < 11:
    print('Smaller')
if x > 21:
    print('Bigger')

print('Finis')
```

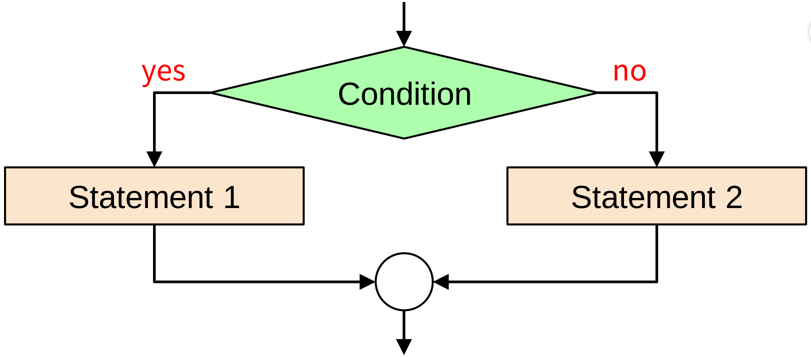
Output:

```
Smaller
Finis
```

Alternative Execution (if-else)

```
if Condition:
    Statement 1
else:

    Statement 2
```

- If the condition **is truthy**:
  - Execute the statement(s) in the if block.
- If the condition **is falsy**:
  - Execute the statement(s) in the else block.



Example: Alternative Execution

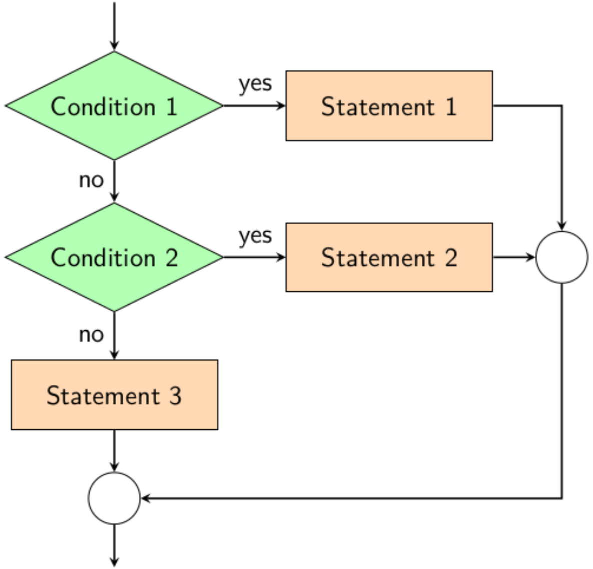
```
# File: tax2.py
income = float(input('Enter income: $'))
if income > 100000:
    tax = 0.4 * income
    income = income - tax
else:
    print('No tax applied.')
print(income)
```

- Using alternative execution, we can extend our tax program to inform the user when no tax is applied.

Chained Conditionals (`if`-`elif`-`else`)

```
if Condition 1:
    Statement 1
elif Condition 2:
    Statement 2
else:
    Statement 3
```

- elif is a contraction of "else if".
- The else clause is optional.
- Multiple elif clauses are allowed.



Example: Chained Conditionals

```
# File: tax3.py
income = float(input('Enter income: $'))
if income > 100000:
    tax = 0.4 * income
    income = income - tax
elif income > 50000:
    tax = 0.3 * income
    income = income - tax

print(income)
```

- Using chained conditionals, we can extend our original tax program to include a 30% tax bracket.

Check Your Understanding

Q. Which print statement will never be executed, regardless of the input?

```python
x = float(input())
if x > 25:
    print('Big')
elif x > 50:
    print('Huge')
elif x > 5:
    print('Medium')
else:
    print('Small')
print('Done.')
```

Check Your Understanding

Q. Which print statement will never be executed, regardless of the input?

A. `print('Huge')`.

Any value of x that could satisfy x > 50 must also satisfy x > 25. Since x > 25 is checked first, print('Huge') can never be executed.

```python
x = float(input())
if x > 25:
    print('Big')
elif x > 50:
    print('Huge')
elif x > 5:
    print('Medium')
else:
    print('Small')
print('Done.')
```

Conditional Execution Pitfalls

Mistake #1: Indentation

- **Indentation** is important---incorrect indentation is a common beginner mistake!

- You must use **consistent indentation**.
  - **I strongly recommend using 4 spaces**.

- Python uses indentation to group statements into blocks.

Indentation Rules

- Increase indentation after **if**, **elif**, and **else** statements.
- Maintain indentation for all statements **grouped** under the **if**, **elif**, or **else** statement.
- Reduce indentation to end the **if**, **elif**, or **else** block.
- **Blank** lines and comment-only lines **ignore indentation** rules.

Example: Incorrect Indentation

```
>>> if x > 2:
...     x = x / 2
... else:
... x = x + 5
  File "<stdin>", line 4
    x = x + 5
    ^

IndentationError: expected an indented block
```

Check Your Understanding

Q. Will this Python script result in an error? If not, what is the expected output?

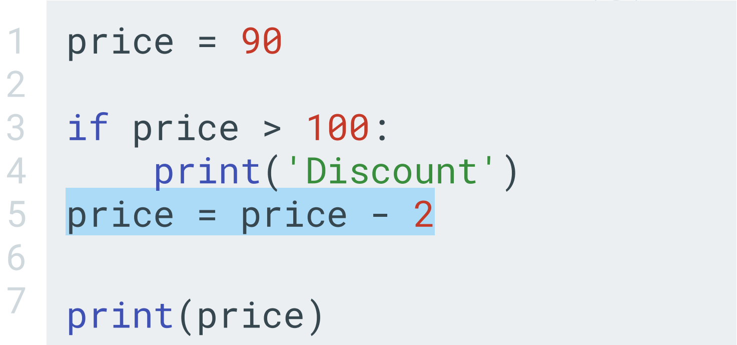
```python
price = 90

if price > 100:
    print('Discount')
price = price - 2


print(price)
```

Check Your Understanding

Q. Will this Python script result in an error? If not, what is the expected output?

A. No error, output 88.

- Line 4 is not executed (condition is not true).
- Lines 5 and 7 are executed, since they are not part of the conditional (indentation reduced).



```
1  price = 90
2
3  if price > 100:
4      print('Discount')
5  price = price - 2
6
7  print(price)
```

Mistake #2: Empty Blocks

- An **if**, **elif**, or **else** statement **must** be followed by at least one indented statement.
  - Comments do not count!

Beware!

The following program will result in an error:

```
x = 5
if x > 10:
    # Do nothing...
else:
    print('Small x')
```

The pass Keyword

- If you really want to do nothing in a block, use the pass keyword.
- In Python, pass is a statement which does nothing.

```
x = 5
if x > 10:
    # Do nothing...
    pass
else:
    print('Small x')
```

Mistake #3: Missing Variable Definitions

- Take care when **creating** variables in a **conditional** and using them afterwards.
- If you create (define) a **variable in one path**, but the control flow does not take that path, the variable will **not** be defined.
    - Trying to use that variable afterwards will result in an error.
- Tip: **before using a variable**, ensure that it is **defined** in all possible control **flows** leading to that point.

Example: Missing Variable Definitions

Beware!

The variable y is not defined, leading to an error.

```
>>> x = 5
>>> if x > 10:
...     y = 42
...
>>> print(x + y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
```

Example: Missing Variable Definitions

- Both of the solutions below ensure that y is always defined before it is used.

```python
x = 5
y = 0

if x > 10:
    y = 42


print(x + y)
```

```python
x = 5
if x > 10:
    y = 42
else:
    y = 0


print(x + y)
```

Nested Conditionals

- Complex decisions can be handled by nesting **if** statements.

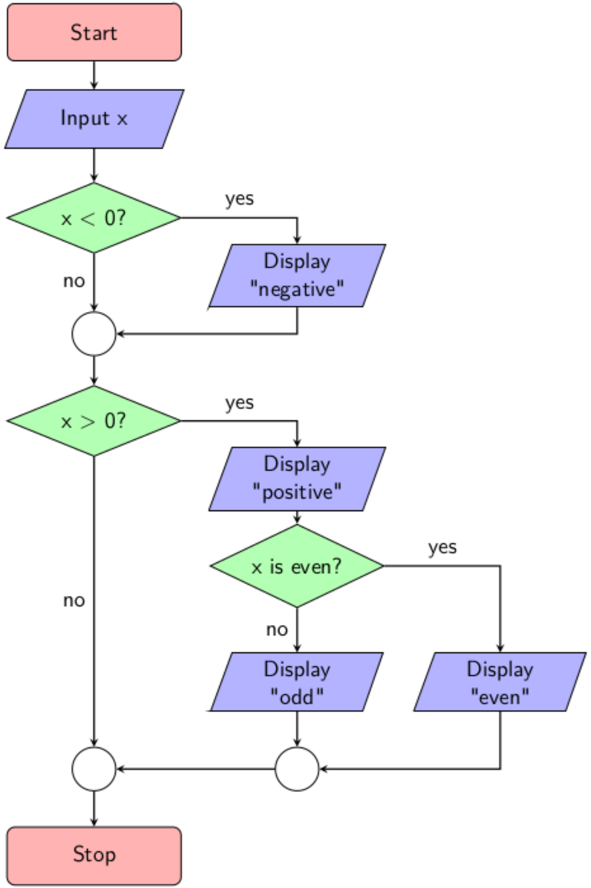
- A nested **if** statement is an **if** statement contained within another ***if***, ***elif***, or ***else*** block.

- In Python, nesting is achieved through **deeper levels** of **indentation**.

Example: Nested Conditionals

Task definition

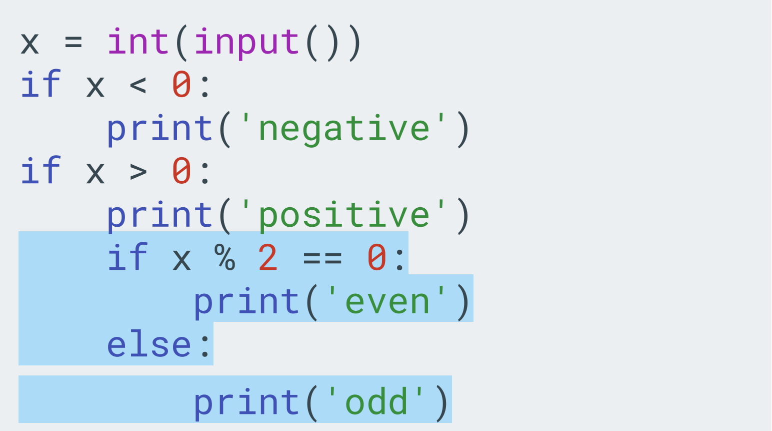
Write a program which prints whether an input number is "negative" or "positive". If the number is positive, also print out whether the number is "even" or "odd".



Example: Nested Conditionals

Task definition

Write a program which prints whether an input number is "negative" or "positive". If the number is positive, also print out whether the number is "even" or "odd".



```
x = int(input())
if x < 0:
    print('negative')
if x > 0:
    print('positive')
    if x % 2 == 0:
        print('even')
    else:

        print('odd')
```

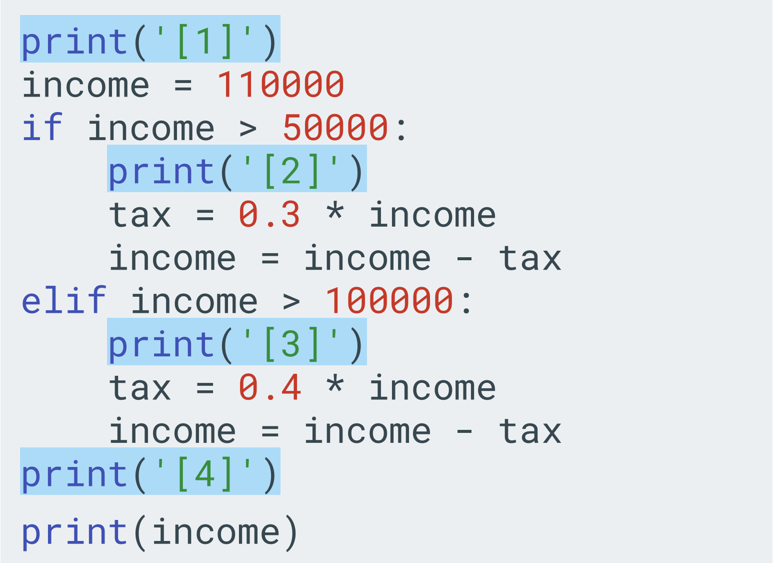
- Then nested if/else statement (highlighted) is only encountered when x > 0 is true.

Things Can Get Confusing!

- Sometimes having a **lot** of **chained** and **nested** conditionals can get confusing.

- Using **print** statements is a good way of tracing control flow.
  - Simply **place** print statements at **various** points in the code.
  - When you don't need the print statements anymore, **delete** them (or **comment** them out).

Example: Tracing Control Flow



```python
print('[1]')
income = 110000
if income > 50000:
    print('[2]')
    tax = 0.3 * income
    income = income - tax
elif income > 100000:
    print('[3]')
    tax = 0.4 * income
    income = income - tax
print('[4]')

print(income)
```

- This program is not working as expected---not enough tax is being applied.

- We can add the highlighted print statements to trace control flow and help us debug the issue.

Example: Tracing Control Flow

```python
print('[1]')
income = 110000
if income > 50000:
    tax = 0.3 * income
    income = income – tax
    print('[2]')
elif income > 100000:
    print('[3]')
    tax = 0.4 * income
    income = income - tax
print('[4]')

print(income)
```

Output:

```
[1]
[2]
[4]

77000.0
```

- We now know which path the control flow takes.
- This also reveals the source of our error: the conditions are checked in the wrong order.

Example: Tracing Control Flow

```python
income = 110000
if income > 100000:
    tax = 0.4 * income
    income = income - tax
elif income > 50000:
    tax = 0.3 * income
    income = income - tax
print(income)
```

- We can fix the problem by reordering the branches in the selection control structure.

- Now that the bug is fixed, we can remove the tracing print statements.

Lecture 2.3

Iteration I

Topic 2.3 Intended Learning Outcomes

- By the end of week 2 you should be able to:
  - Use iteration control structures to repeat the execution of statements,
  - Use a Python range to specify a sequence of numbers, and
  - Perform aggregations like finding the average or maximum value of a sequence.

Lecture Overview

1. While Loops
2. Kinds of While Loops
3. Nested Loops and Finishing Early

While Loops

Repeated Actions

- Often times you will find that programs **require** the same **action** to be performed **multiple** times.

- The data might **change**, but the **fundamental** action is the **same**.

Example: Times Table

Task Definition

Print the times table for a number input by the user. So if the user inputs 7, the output should be:

- 7 × 1 = 7
- …
- 7 × 12 = 84

```
a = int(input('Times table: '))
print(str(a) + ‘ * 1 = ' + str(a * 1))
print(str(a) + ‘ * 2 = ' + str(a * 2))
print(str(a) + ‘ * 3 = ' + str(a * 3))
print(str(a) + ‘ * 4 = ' + str(a * 4))
print(str(a) + ‘ * 5 = ' + str(a * 5))
print(str(a) + ‘ * 6 = ' + str(a * 6))
print(str(a) + ‘ * 7 = ' + str(a * 7))
print(str(a) + ‘ * 8 = ' + str(a * 8))
print(str(a) + ‘ * 9 = ' + str(a * 9))
print(str(a) + ‘ * 10 = ' + str(a * 10))
print(str(a) + ‘ * 11 = ' + str(a * 11))
print(str(a) + ‘ * 12 = ' + str(a * 12))
```

- The above code works, but is highly redundant!

Repeated Actions

Q: Write a python code to print 1 to 5.

Python Code: Print 1 to 5

```
print (1)
print (2)
print (3)
print (4)
print (5)
```

- How about printing 1 to 10,000 or to 100,000?
- Are you going to write or copy, paste, and modify?
- Yes - it could be but is it practical?
- **Repetitive Execution** can help us to solve this issue.

Redundant Code is Bad

- Having a lot of similar code repeated causes a few **problems**.
    - **Time-consuming** to write.
    - **Changing** the code is **error-prone**.
    - **Inflexible** (what if you wanted to show up to 20x).
- Fortunately there is a better way using iteration.

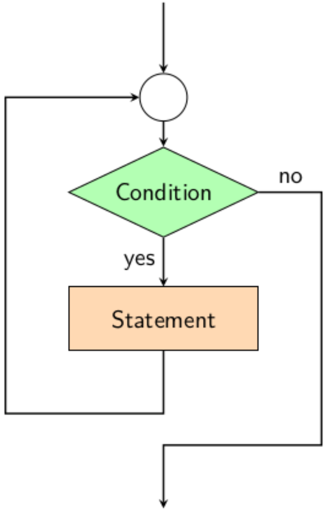
Repetitive Execution: Iteration

- In an iteration control structure, a condition determines **how many times statements are executed**.

- **Allows** the program to execute the **same statement(s) multiple** times.

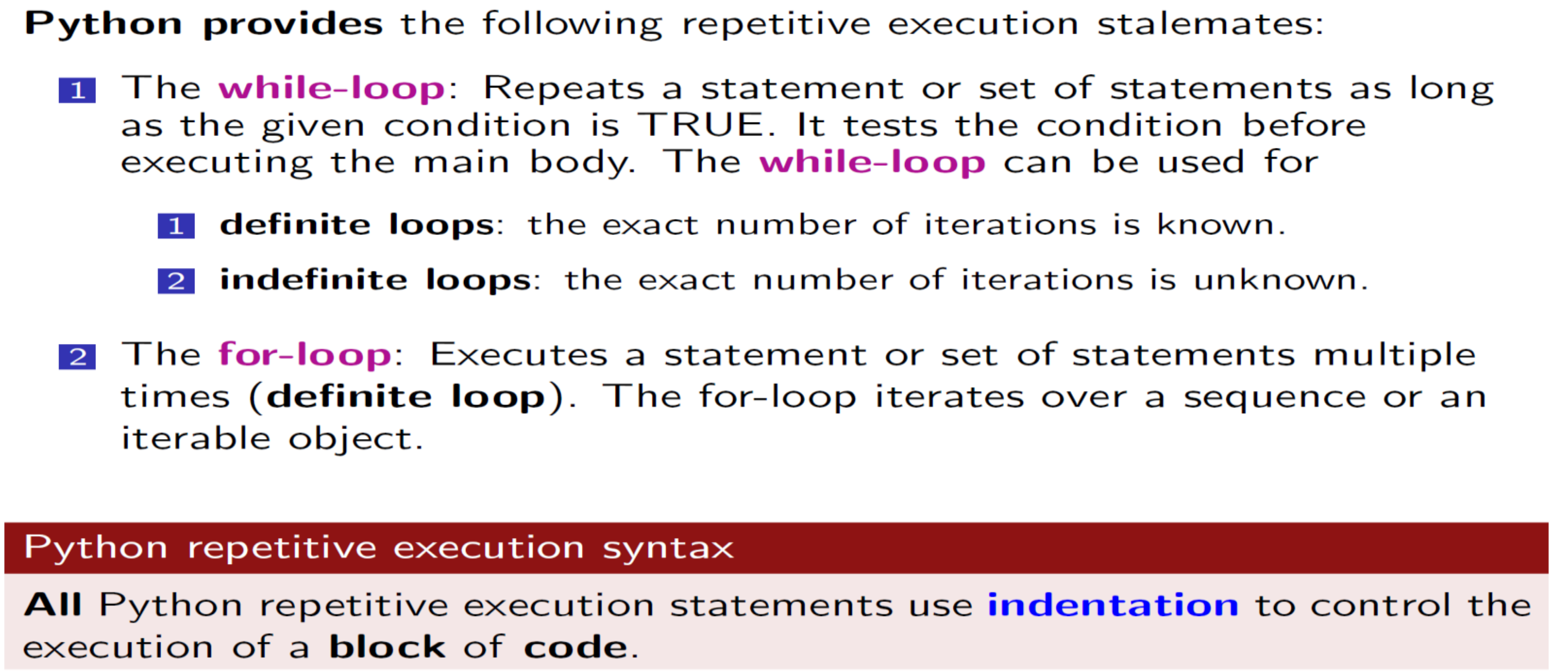
- **Achieved** by **allowing** control flow to **return to an earlier point** in the program.

- Represented by a **backwards-pointing** arrow in a flowchart.



Repetitive Execution: Iteration

Repetition: Repeating a set of actions for a number of times.



Python provides the following repetitive execution stalemates:

1. The **while-loop**: Repeats a statement or set of statements as long as the given condition is TRUE. It tests the condition before executing the main body. The **while-loop** can be used for
    1. **definite loops**: the exact number of iterations is known.
    2. **indefinite loops**: the exact number of iterations is unknown.
2. The **for-loop**: Executes a statement or set of statements multiple times (**definite loop**). The for-loop iterates over a sequence or an iterable object.

Python repetitive execution syntax

All Python repetitive execution statements use **indentation** to control the execution of a **block** of **code**.

While Loops

while-loop

A while-loop is condition-controlled statement that uses a condition to control the repetitive execution of a **block** of **code**.

Python uses the `while` keyword for the **while-loop statement**:

Pseudo Code: while-loop

```
while (condition):
    statement
    statement
    etc.
```

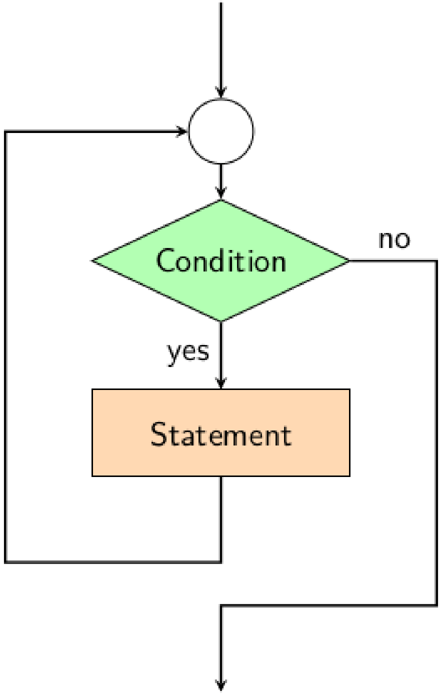
From the above we can see that

- a while-loop statement is made up of `while` keyword, followed by the **condition** and **colon** (:) at the end.
- there is an **indentation** which means the statements will execute as long as the condition is satisfied

While Loops

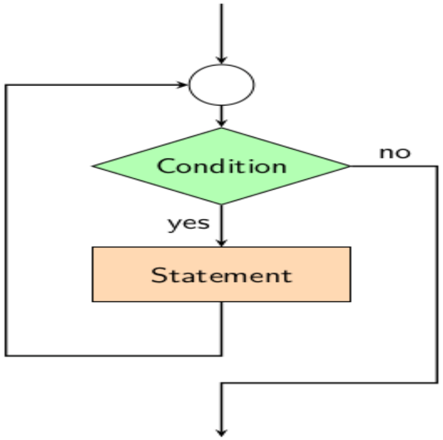
```
while Condition:
    Statement
```

- If the condition **is truthy**:
  - Execute the statement(s) in the while block.
  - Check the condition again.
- If the condition **is falsy**:
  - Exit the while loop.



While Loops

```
i = 1
while i <= 4:
     print (i)

     i=i+1

Print (“Done!”)

# output

1

2

3

4

Done!
```



```
i = 1 # ←---- initialisation
while i <= 4: # <--- test (Boolean test)
     print (i) # ←--- action-during
     i=i+1      # ←---  each-loop
Print (“Done!”)
```

While loops

The while statement keeps looping until the conditional evaluates to False. The **condition** should be updated and checked every time we call the while statement.

Example (while-loop: definite loop)

```
>>> num = 6
>>> while (num > 0): # check condition
...     print(num)
...     num=num-1 # update the condition counter
...
6
5
4
3
2
1
>>> print ("End of while-loop")
End of while-loop
```

Example: re-write the Times Table code using **While** Loops

Task Definition

Print the times table for a number input by the user. So if the user inputs 7, the output should be:

- 7 × 1 = 7
- …
- 7 × 12 = 84

```
a = int(input('Times table: '))
print(str(a) + ‘ * 1 = ' + str(a * 1))
print(str(a) + ‘ * 2 = ' + str(a * 2))
print(str(a) + ‘ * 3 = ' + str(a * 3))
print(str(a) + ‘ * 4 = ' + str(a * 4))
print(str(a) + ‘ * 5 = ' + str(a * 5))
print(str(a) + ‘ * 6 = ' + str(a * 6))
print(str(a) + ‘ * 7 = ' + str(a * 7))
print(str(a) + ‘ * 8 = ' + str(a * 8))
print(str(a) + ‘ * 9 = ' + str(a * 9))
print(str(a) + ‘ * 10 = ' + str(a * 10))
print(str(a) + ‘ * 11 = ' + str(a * 11))
print(str(a) + ‘ * 12 = ' + str(a * 12))
```

- The above code works, but is highly redundant!

Example: re-write the Times Table code using **While** Loops

```
a = int(input('Times table: '))
b = 1 # initialisation
while b <= 12: # check condition
 print(str(a) + ' x ' + str(b) + ' = ' + str(a * b))
 b = b + 1 # update condition
```

- **Same output** as the earlier **repetitive** code.
- Only requires **one *print*** statement.
- **b** is a counter variable which keeps track of how many times the loop has **repeated**.
- When b **exceeds** 12, the loop **exits**.

Check Your Understanding

Q. How many times will the program print output?

```
x = 5
while x >= 0:
    print(x)
    x = x - 1
```

Check Your Understanding

Q. How many times will the program print output?

```
x = 5
while x >= 0:
    print(x)
    x = x - 1
```

A. Six times.

The output will be:

```
5
4
3
2
1
0
```

Kinds of While Loops

Definite While Loops

- A definite loop repeats a **fixed** number of **times**.
  - The number of repetitions is **known** before starting the loop.

- The improved "**times table**" example from earlier used a **definite** loop (it **looped 12 times**).

Indefinite While Loops

- An indefinite loop repeats for a number of times which is **not obvious before** starting the loop.

- An indefinite loop could:
  - **Finish** based on **user input** (interactive loop).
  - **Finish** based on a **computed value**.
  - **Never finish** (infinite loop).

Indefinite While Loops: **Interactive Loops**

- An interactive loop has a **condition** which **decides** when to **finish** repeating based on **user input**.

- Hence the user **controls** when the loop **finishes** *as it is running*.

Example: Total Cost Calculator

```
total_cost = 0
item_cost = float(input('Item cost: '))
while item_cost >= 0:
    total_cost = total_cost + item_cost
    item_cost = float(input('Item cost: '))

print(total_cost)
```

- This program will run **until** the user enters a **negative** number.
- The **user** can **decide** how many **items** they want to add.

Computed Conditions

- A loop can have a **condition** which is **based** on a **complex computation** made by the **loop**.

- The **number** of **repetitions** is **not** obvious before **entering** the **loop**.

- Can be **used** for certain **mathematical calculations**.

Example: Collatz Conjecture

- The "Collatz conjecture" is a famous mathematical problem:
  - Consider a **positive integer** *n*.
  - If the number is even, halve it.
  - If the number is odd, triple it and add **1**.
  - The **conjecture** is that if you **repeat** this enough **times**, you will always **reach** the **number 1**.

- The number of iterations required to **reach 1** is called the "**total stopping time**".

Example: Collatz Conjecture

Task Definition

Write a program which calculates the total stopping time (as defined by the Collatz conjecture) for a number input by the user.

```
n = int(input('Enter n: '))
t = 0
# Repeat until n reaches 1.
while n != 1:
  if n % 2 == 0:
    # n is even.
    n = n / 2
  else:
    # n is odd.
    n = n * 3 + 1
  # Increment total stopping time.
  t = t + 1

print(t)
```

Example: Collatz Conjecture

- The **calculation** in the loop itself **determines** how **many** times it will repeat.
  - This is what makes it **indefinite**.

- Fun aside: if you can somehow **figure out** the total **stopping** time without a loop, you may be well on your way to solving the Collatz conjecture!

```
n = int(input('Enter n: '))
t = 0
# Repeat until n reaches 1.
while n != 1:
  if n % 2 == 0:
    # n is even.
    n = n / 2
  else:
    # n is odd.
    n = n * 3 + 1
  # Increment total stopping time.
  t = t + 1

print(t)
```

Indefinite While Loops: **Infinite** Loops

- An infinite loop **never finishes** (it **repeats forever**).
    - In practice this means **until** the program **crashes** or is **forced** to **quit** (e.g. shutting down).

- This occurs when the **condition** is always **truthy**.

- If you accidentally run a program containing an infinite loop in a script or interactive session, you can force the program to **exit** by pressing **Ctrl+C**.

Check Your Understanding

Q. How many times will the Python script print output?

```
x = 1
while x * 2 < 10:
    print(x)
x = x + 1
```

Check Your Understanding

Q. How many times will the Python script print output?

A. Infinite times.

- The last line is not part of the while loop since it is de-indented.
- Since x stays at its initial value of 1. the condition is always true.

```
x = 1
while x * 2 < 10:
    print(x)
x = x + 1
```

Examples: While loops

Python Code: while-loop: indefinite loop

```
f = 1
while (f != 0): # check condition
    print ("Hi")
    print ("To stop the program enter 0") # update the condition
    f = int(input("Enter a number  :"))
    print ("You entered: ", f)

print ("Terminated")
```

Python Code: while-loop: definite loop - sum of 1 to 10

```
n=10
sum=0
i=0
while (i < n): # check condition
    sum=sum+i
    i=i+1 # update the condition
print ("The sum of 1 to 10 is {}".format(sum))
```

Python Code: while-loop: indefinite loop

```
message = ""
while (message != 'exit'): # check condition
    message = input("enter a message or exit to stop: ")
    print(message)
```

Nested Loops and Finishing Early

Nested Loops

- Like if **statements**, **while loops** can be **nested**.

- Nesting loops **multiplies** the **number** of times that the **innermost** loop is **repeated**.
  - Programs with a lot of **nested** loops can end up running quite **slowly** as a result.

Example: All Times Tables

```python
a = 1
while a <= 12:
    b = 1
    while b <= 12:
        print(str(a) + ' x ' + str(b) + ' = ' + str(a * b))
        b = b + 1
    a = a + 1
```

- **Highlighted** lines are **copied** from **earlier**.
  - e.g. when a = 7, this part prints the 7 times table.
- An outer loop has been added.
  - This loops **through** a = 1, 2, 3, . . ., 12

Finishing the Current **Iteration**

- In Python, a continue statement can be used to finish the **current iteration early**.
  - Written using the continue keyword.

- **Control flow** will immediately **return** to the ***while loop's*** condition.

Finishing the Current **Iteration**

The continue statement: the while statement keeps looping and skips a loop when the continue conditional is True.

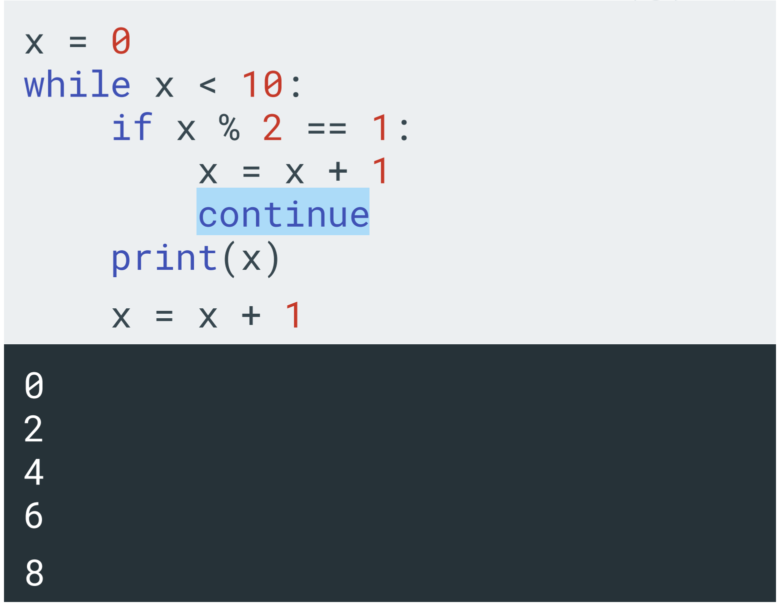
Example (continue statement)

```
>>> num = 6
>>> while num > 0:
...     num -= 1
...     if num == 2:
...         continue
...     print(num)
...
5
4
3
1
0
```

Example: Printing Even Numbers

- For each repetition where x % 2 == 1 (i.e. **x** is **odd**), the continue statement will be **reached**.

- The **continue** statement returns to the **top** of the **while loop** immediately.
    - Hence the print statement is **skipped**.



```
x = 0
while x < 10:
    if x % 2 == 1:
        x = x + 1
        continue
    print(x)
    x = x + 1
```

```
0
2
4
6
8
```

Finishing the **Entire** Loop

- In Python, a break statement can be used to **finish** the entire **loop** early.
    - Written using the break keyword.

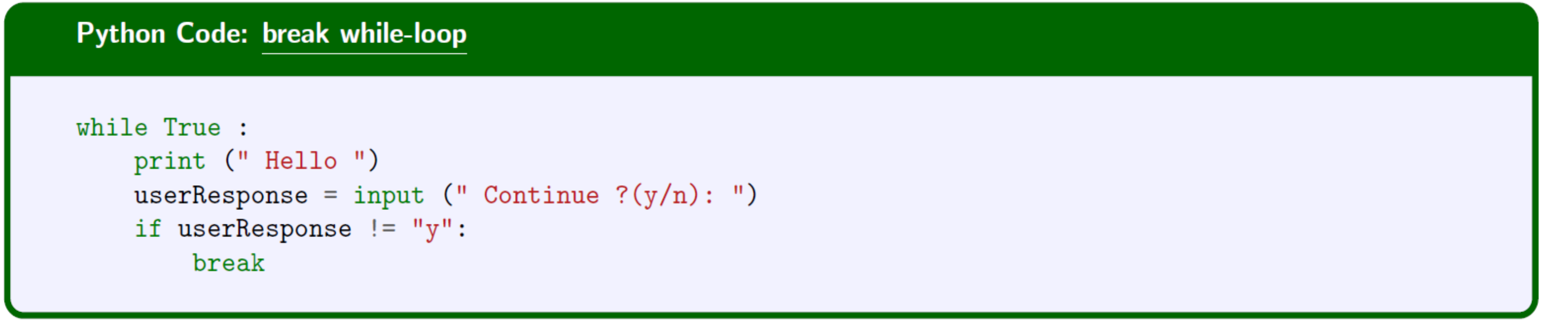
- **Control flow** will immediately **skip** to the code after the **loop**.

Finishing the Entire Loop

The `while` statement keeps looping until the `break` conditional is `True`.

Python Code: break while-loop

```
>>> num = 4
>>> while num > 0:
...     num -= 1
...     print(num)
...     if num == 2:
...         break
...
...
...
3
2
>>>
```



Python Code: break while-loop

```
while True :
    print (" Hello ")
    userResponse = input (" Continue ?(y/n): ")
    if userResponse != "y":
        break
```

Example: Total Cost Calculator

```python
total_cost = 0
item_cost = float(input('Item cost: '))
while item_cost >= 0:
    total_cost = total_cost + item_cost
    item_cost = float(input('Item cost: '))
print(total_cost)
```

- Our previous implementation (shown) has duplicate code.

Example: Total Cost Calculator

```
total_cost = 0
while True:
    item_cost = float(input('Item cost: '))
    if item_cost < 0:
        break
    total_cost = total_cost + item_cost

print(total_cost)
```

- By adding a conditional **break** statement (highlighted), we can **avoid** the **duplication**.
- It is now the **if statement** which **controls** when the loop **finishes**.

Check Your Understanding

Q. How many times will the script print output?

```
x = 0
while x < 5:
    if x % 2 == 0:
        x = x + 1
        continue
    y = 0
    while y < x:
        print(y)
        y = y + 1
    x = x + 1
```

Check Your Understanding

Q. How many times will the script print output?

A. 4 times.

- The outer loop repeats 5 times (x=0,1,2,3,4).
- Three of the outer loop repetitions exit early (x=0,2,4).
- When x = 1, the inner loop prints 1 time.
- When x = 3, the inner loop prints 3 times.

```
x = 0
while x < 5:
    if x % 2 == 0:
        x = x + 1
        continue
    y = 0
    while y < x:
        print(y)
        y = y + 1
    x = x + 1
```

Lecture 2.4

Iteration II

Lecture Overview

1. For Loops
2. Example Program: FizzBuzz
3. Aggregation

For Loops

For Loops

The `for` statement loops through all the elements in a data container. In `for` loop, we need to know how many times the block of code will be repeated.

Pseudo Code: for-lop

```
for each element in sequence:
    # work on element
```

From the above we can see that

- a for-loop statement is made up of `for` and `in` keywords, followed by the **sequence** and **colon** (:) at the end.
- there is an **indentation** which means the statements will execute till the end of the given sequence.

While Loops vs. For Loops

- Previously we saw how while loops can be used to perform iteration in Python.
  - The while loop is **controlled** by a **condition**.

- A common use for iteration is to perform an **action** for each item in a sequence.
  - e.g. Add numbers from 1 to 10, print all customers.

- Python provides for loops as a more **convenient** way of iterating **over** a **sequence**.

While Loops vs. For Loops

```
# While loop
i = 0
while i < 10:
    print(i)
    i = i + 1
```

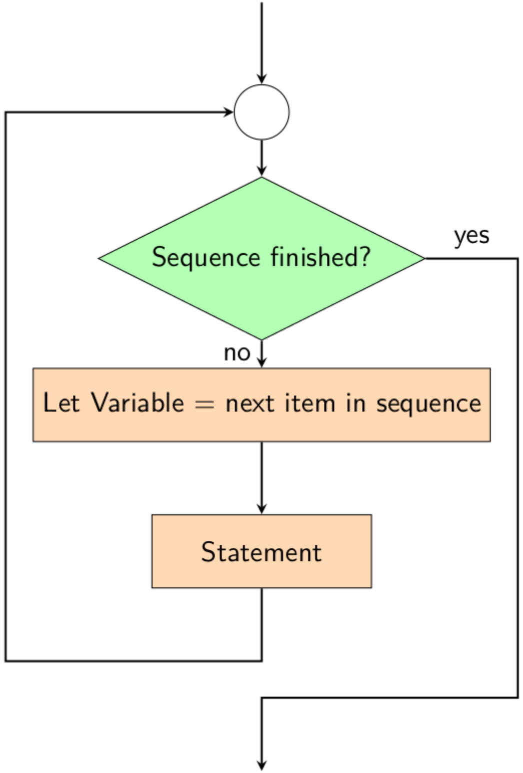
```
# For loop
for i in range(10):
    print(i)
```

- Both code snippets print numbers from 0 to 9.
- The for loop version is better.
    - More concise.
    - Easier to read (once we learn about range).
    - Less error prone (can't forget to increment i).

For Loops

```
for Variable in Sequence:
    Statement
```

- While there are **items left** in the **sequence**:
  - Assign the **next** item in the **sequence** to the **iteration** variable.
  - **Execute** the **statements** in the for block.



For Loops

- For loops share a lot in common with while loops.
  - You can **use** continue and break statements to finish early.
  - You can **nest** them.
  - You **must** follow the same **indentation rules**.

For Loops

Example: Use for loop to print 1 to 5

Example (Print 1 to 5)

```
>>> for i in 1,2,3,4,5:
...     print(i)
...
1
2
3
4
5
```

- How about printing 1 to 100?
- Is it practical to list all items after `in`?
- Python provides a useful function known as the `range()` function to specify how many times the for-loop will repeat.

For Loops: range () function

We can use the range() function to specify the number of times the for-loop will repeat.

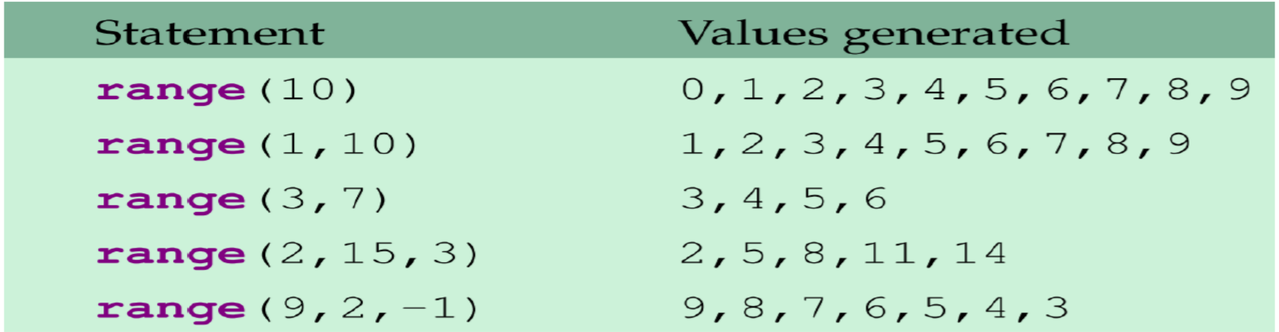
Pseudo Code: for-loop: range()

```
for variable_name in range(number of times to repeat):
    # statements_to_be_repeated
```

- The value we add into the range() function determines how many times the for-loop will repeat.
- The range() function produces a list of numbers from zero to the value minus one.
- For instance, range(4) produces four values: 0, 1, 2, and 3.

For Loops- range () function

- The general form of the range expression is: range (**begin**, **end**, **step**) where
  - **begin**: the first value in the range; if not provided, the default value is **0**.
  - **end**: is the last value minus one (n-1) in the range and should be provided.
  - **step**: is the amount to increment or decrement. if not provided, it defaults is **1**.



Statement	Values generated
`range(10)`	0,1,2,3,4,5,6,7,8,9
`range(1,10)`	1,2,3,4,5,6,7,8,9
`range(3,7)`	3,4,5,6
`range(2,15,3)`	2,5,8,11,14
`range(9,2,−1)`	9,8,7,6,5,4,3

Writing a Python Range

- Let's say you have a **sequence** in mind that you want to **express** using a Python **range**.

- Firstly you need to check whether this is **possible**:
  - Are all of the numbers unique integers?
  - Is the sequence an arithmetic sequence (i.e. do items in the sequence **increase/decrease** by a **constant** amount)?

Valid Range Sequences

Valid Range Sequences

- ✓ 1, 2, 3, 4, 5
- ✓ 8, 10, 12
- ✓ 6, 3, 0, -3
- ✓ 100

Invalid Range Sequences

- ✗ 1, 4, 9, 16, 25
- ✗ 1.1, 2.1, 3.1
- ✗ 5, 5, 5, 5

Steps For Writing a Python Range

1. Think of the first number in the sequence. This is *m*.
2. Think of the difference between the second and first item in the sequence. This is *step*.
3. Think of the **last** number in the **sequence**, then **add step** to it. This is *n*.

The Python code for the sequence is:

```
range(m, n, step)
```

Simplifying Python Ranges

- In some instances you can simplify the range. `range(m, n, step)`
  - If **step** is **1**, you can **omit** it.
  - If step is **1** *and* ***m*** is 0, you can **omit** them both.
- For example:
  - `range(6, 9, 1)` simplifies to `range(6, 9)`
  - `range(0, 3, 1)` simplifies to `range(3)`

Examples: Writing a Python Range

- 0, 1, 2, 3, 4
  - `range(0, 5, 1)`, or simply `range(5)`
- 1, 2, 3
  - `range(1, 4, 1)`, or simply `range(1, 4)`
- 0, 2, 4, …, t
  - `range (0, t + 2, 2)`
- y, y-1, y-2, …, y-x
  - `range(y, (y-x)-1, -1)`

Examples: For Loops:

Example

```
>>> for i in range(4):
...     print('*'*5)
...
*****
*****
*****
*****
```

Example

```
>>> for i in range(3):
...     print('*'*(i+1))
...
*
**
***
```

Examples: For Loops

Example: Write a python program to find all numbers which are divisible by 5 but are not a multiple of 3, between 10 and 50

Python Code:

```
>>> for i in range(10, 50):
...     if (i%5==0) and (i%3!=0):
...         print (i, " ", end="")
...
10  20  25  35  40
```

Example: Write a python program to counts how many of the squares of the numbers in 1 to 100 end with digit 1 or 9.

Python Code:

```
>>> count = 0
>>> for i in range(1,100):
...     if (i**i % 10 == 1 or i**i % 10 == 9):
...         count += 1
...         print(i, " ", end="")
...
1  9  11  19  21  29  31  39  41  49  51  59  61  69  71  79  81  89  91  99
>>> print("count: ", count)
count:  20
```

Examples: For Loops

Example: Write a Python program print power 10 (10^{1to10}) for 1 to 10

Python Code: for-loop

```
>>> for i in range(10):
...     print('{0:3} {1:10}'.format(i, 10**i))
...
  0          1
  1         10
  2        100
  3       1000
  4      10000
  5     100000
  6    1000000
  7   10000000
  8  100000000
  9 1000000000
```

Check Your Understanding

Q. Fill in the blank using a Python range such that the program output is 9, 8, 7, 6.

```
for i in ANSWER:
    print(i)
```

Check Your Understanding

Q. Fill in the blank using a Python range such that the program output is 9, 8, 7, 6.

```
for i in ANSWER:
    print(i)
```

A. `range(9, 5, -1)`

1. The first value is 9.
   - ➢ m = 9
2. The first two values are 9, 8.
   - ➢ step = 8 - 9 = -1
3. The last value is 6.
   - ➢ n = 6 + step = 5

Example Program: FizzBuzz

What is FizzBuzz?

- FizzBuzz is a programming problem often used during software developer job interviews.
- Tests knowledge of:
  - **Selection** and **iteration** control structures,
  - **Boolean** expressions, and
  - The **modulo** operator.
- We have learnt about each of these things, so we are now equipped to tackle FizzBuzz!

Task Definition

Task Definition

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

Source: https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/

Example Output

- The first 16 lines of the expected program output are shown here.

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16

...
```

Printing Numbers from 1 to 100

- To begin with, let's ignore the fizzing/buzzing and focus on simply printing numbers from 1 to 100.

- We know that we can iterate over these numbers using a **for loop** and a Python **range**.

- How can we express the sequence of numbers from 1 to 100 (inclusive) using a Python range?

```
range(1, 101)
```

Printing Numbers from 1 to 100

```
for i in range(1, 101):
    print(i)
```

- The above code will print all integers from 1 to 100, inclusive.

Fizzing and Buzzing

- Let's now consider the full task definition:

*Write a program that **prints the numbers** from 1 to 100. But for multiples of three **print "Fizz"** instead of the number and for the multiples of five **print "Buzz"**. For numbers which are multiples of both three and five **print "FizzBuzz"**.*

- We can identify that the program should select one of four different outputs for each number.
  - The number itself (this is the "default" option).
  - "Fizz" (multiple of 3).
  - "Buzz" (multiple of 5).
  - "FizzBuzz" (multiple of 3 *and* 5).

Fizzing and Buzzing

- We will need to use a selection structure with four branches.

```
for i in range(1, 101):
    if Condition 1:
        print(Output 1)
    elif Condition 2:
        print(Output 2)
    elif Condition 3:
        print(Output 3)
    else:
        print(i) # The "default" option.
```

Fizzing and Buzzing

- We are testing conditions based on whether the number is a multiple of other numbers (3 and/or 5).

- Some number **x** is a multiple of another number **y** if and only if **x** can be divided by **y** with **no remainder**.
  - Recall that the modulo operator calculates the remainder.
  - Hence **x** is a multiple of **y** if and only if `x % y == 0`.

Fizzing and Buzzing

- Let's pair each output with its condition:
  - *multiples of three print "Fizz"*
    - Condition: `i % 3 == 0`
    - Output: "Fizz“
  - *multiples of five print "Buzz"*
    - Condition: `i % 5 == 0`
    - Output: "Buzz“
  - *multiples of both three and five print "FizzBuzz"*
    - Condition: `i % 3 == 0 and i % 5 == 0`
    - Output: "FizzBuzz"

First Solution Attempt (Incorrect)

```
for i in range(1, 101):
    if i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    elif i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    else:
        print(i)
```

- Can you spot the bug in the code?

First Solution Attempt (Incorrect)

- The conditions are tested **in order of appearance**.
- Any value of **i** which satisfies the "FizzBuzz" condition will satisfy the "Fizz" condition.
- Since the "Fizz" condition is tested first, the program will output "Fizz" instead of "FizzBuzz".

```python
for i in range(1, 101):
    if i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    elif i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    else:

        print(i)
```

Second Solution Attempt (Correct)

- The bug can be fixed by reordering the branches.
- Importantly the most specific condition is tested first.

```
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

Check Your Understanding

Q. Does the order of the two elif branches matter for getting the correct output?

```
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

Check Your Understanding

Q. Does the order of the two elif branches matter for getting the correct output?

A. No.

A value of **i** which does not satisfy the FizzBuzz condition can never satisfy both the Fizz and Buzz conditions. Therefore the ordering of these conditions does not matter.

```python
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

3.
Aggregation

Aggregation

- Aggregation is the process of **combining** values to **produce** a **single** summary **value**.

- Some common **aggregations** include:
  - **Sum**,
  - **Average** (**mean**), and
  - **Maximum** value.
- Using a **for loop** is a convenient way of **aggregating**.

Aggregation Pattern

- A general pattern for aggregation is:
  1. Define one (or more) summary **variables**, using suitable **initial values**.
  2. Write a **for loop** to update the **summary** variable(s) using each item in the **sequence**.
  3. Use the **summary** variable(s) to calculate the **final** result.
- **Step 3** is not always necessary---in some cases the summary variable itself is the result.

Aggregation: Sum

- This program sums the numbers from 1 to 9.

- **Summary** variable: `total`.

- We keep track of the running total as each item in the sequence is considered.

```python
total = 0
for x in range(1, 10):
    total = total + x
print(total)
```

Aggregation: Average (Mean)

- This program averages the numbers from 1 to 9.
- Summary variables: `total` and `count`.
- We keep track of both the total sum and the count, then calculate the average at the end by dividing.

```
total = 0
count = 0
for x in range(1, 10):
    total = total + x
    count = count + 1
average = total / count
print(average)
```

Aggregation: Maximum

- This program finds the maximum value from a list.
  - More about lists in a future lecture.
- Summary variable: `maximum`.
- An **if statement** is used to conditionally **update** `maximum` whenever a larger value is encountered.

```python
maximum = 0
for x in [3, 1, 8, 4]:
    if x > maximum:
        maximum = x

print(maximum)
```

Count

- This program counts the number of values greater than 5 in a list.

- Summary variable: `count`.

- The count is incremented for each item which meets the if statement condition.

```
count = 0
for x in [4, 9, 5, 8, 9]:
    if x > 5:
        count = count + 1

print(count)
```

Custom Aggregation

- With a little bit of thinking you can devise your own **custom aggregations**.

- You need to:
  1. **Define** the sequence to aggregate.
  2. **Identify** summary variable(s).
  3. **Identify** the repeated operation(s) which builds up the summary variable(s).
  4. Use the **summary** variable(s) to calculate the final result (not always necessary).

Custom Aggregation

Task Definition

Write a program which concatenates all single-digit numbers into one long string surrounded by square brackets and prints the result.

1. The sequence we are iterating over is 0, 1, …, 9.
2. The **summary variable** is the **string** being built.
3. The repeated operation is **converting** the **item** to a **string** and **concatenating** it with the **summary** variable.
4. The final result is the summary variable in **square brackets**.

Custom Aggregation

```
s = ''
for x in range(10):
    s = s + str(x)
print('[‘ + s + ']’)

#output

>>> [0123456789]
```

1. The sequence we are iterating over is 0, 1, …, 9.
2. The summary variable is the string being built.
3. The repeated operation is converting the item to a string and concatenating it with the summary variable.
4. The final result is the summary variable in square brackets.

Custom Aggregation

- If you are still unsure about how this code works, think about what it looks like with the loop "unrolled".

- Recall that the str function converts a value to a string.
  - e.g. `str(0)` gives `'0'`

```
s = ''
for x in range(10):
    s = s + str(x)
print('[‘ + s + ']')
```

```
s = ''
s = s + str(0)
s = s + str(1)
...
s = s + str(9)
print('[‘ + s + ']')
```

Check Your Understanding

Q. Which program correctly calculates the product of values between 1 and 10 (inclusive)?

```
# Program A.
prod = 1
for x in range(1, 11):
    prod = prod * x
print(prod)

# Program B.
prod = 0
for x in range(1, 11):
    prod = prod * x
print(prod)

# Program C.
prod = 1
for x in range(1, 11):
    x = prod * x

print(prod)
```

Check Your Understanding

Q. Which program correctly calculates the *product* of values between 1 and 10 (inclusive)?

A. Program A.

- Program B uses an inappropriate initial value and will output 0.
- Program C does not update the summary variable.

```
# Program A.
prod = 1
for x in range(1, 11):
    prod = prod * x
print(prod)

# Program B.
prod = 0
for x in range(1, 11):
    prod = prod * x
print(prod)

# Program C.
prod = 1
for x in range(1, 11):
    x = prod * x

print(prod)
```

Next Lecture We Will...

- Discover how functions allow us to use and write reusable chunks of code.
- Leanr how to use objects and implement string and files

Thanks for your attention!

The slides and lecture recording will be made available on LMS.

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