CONDITIONS

PREPARED BY: LUIS MEING

OBJECTIVES:

01 **Definition**

02 **Introduction to Boolean Algebra**

2.1: History and Meaning
2.2: Recap on Prescedence
2.3: Conjunction
2.4: Disjunction
2.5: Negation

03 **Decision Control Structures**

3.1: Definition
3.2: ‘if’ Statement
3.3: ‘if … else’ Statement
3.4: ‘if … elif’ Statement
3.5: ‘if…elif…else’ Statement
3.6: ‘Match-Case’ Statement

04 **Nesting**

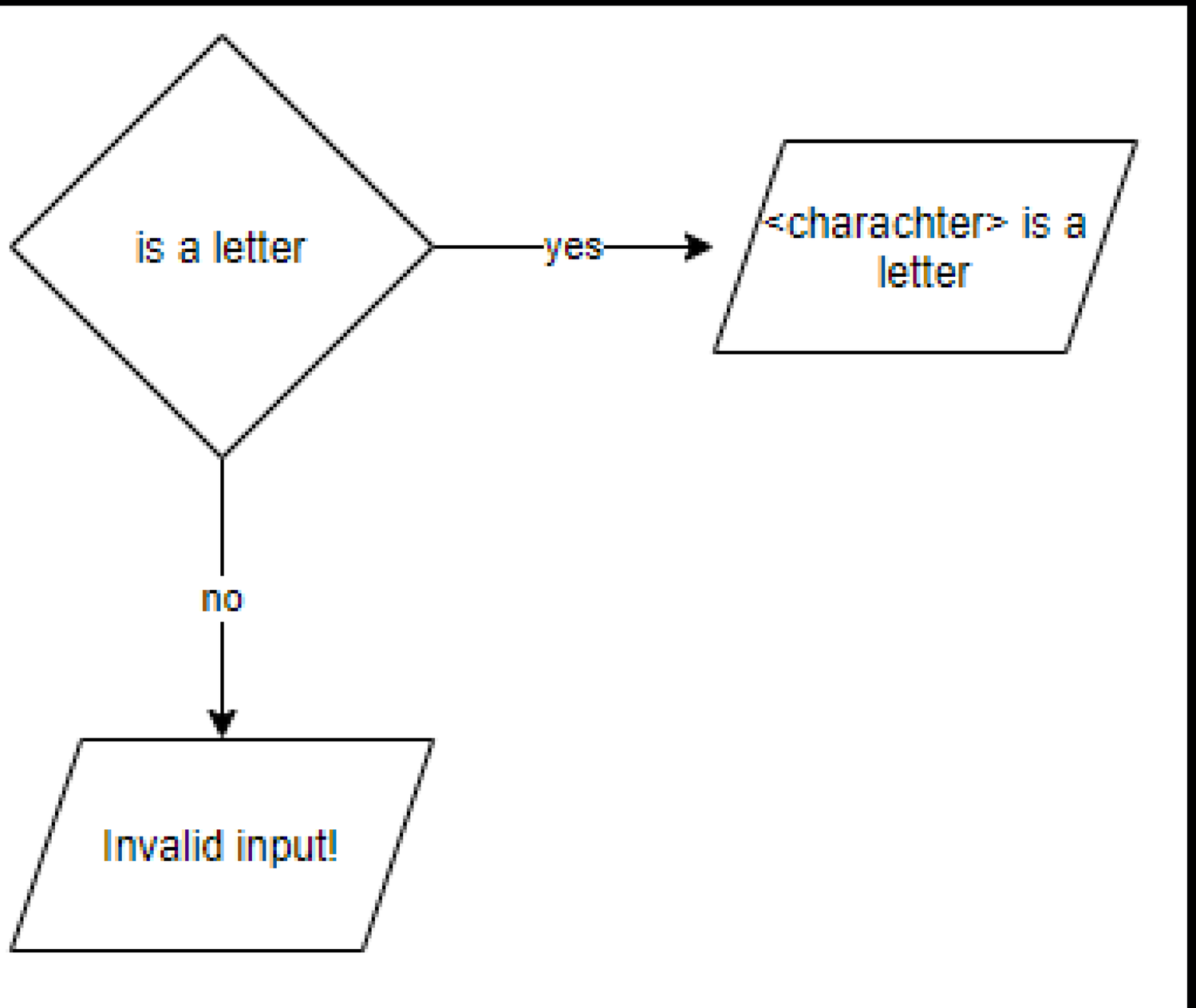
Definition

- A condition is something that a computer can decide is either true or false.True is like the computer is answering yes and false is like answering no. You can tell your app to do different things depending on if the condition is true or false.

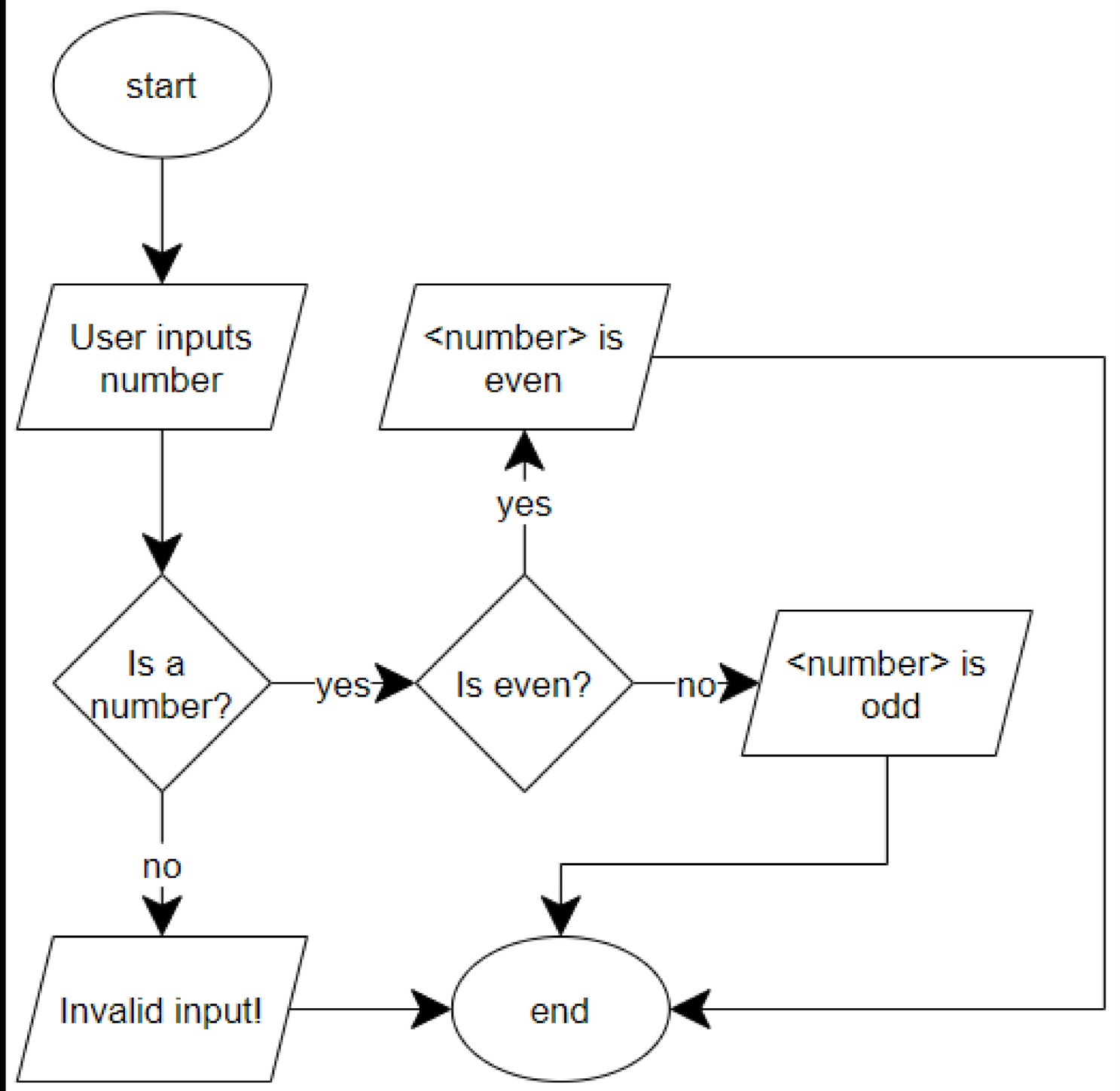
Definition

- These are branches of the algorithm, the logic of where your code is going

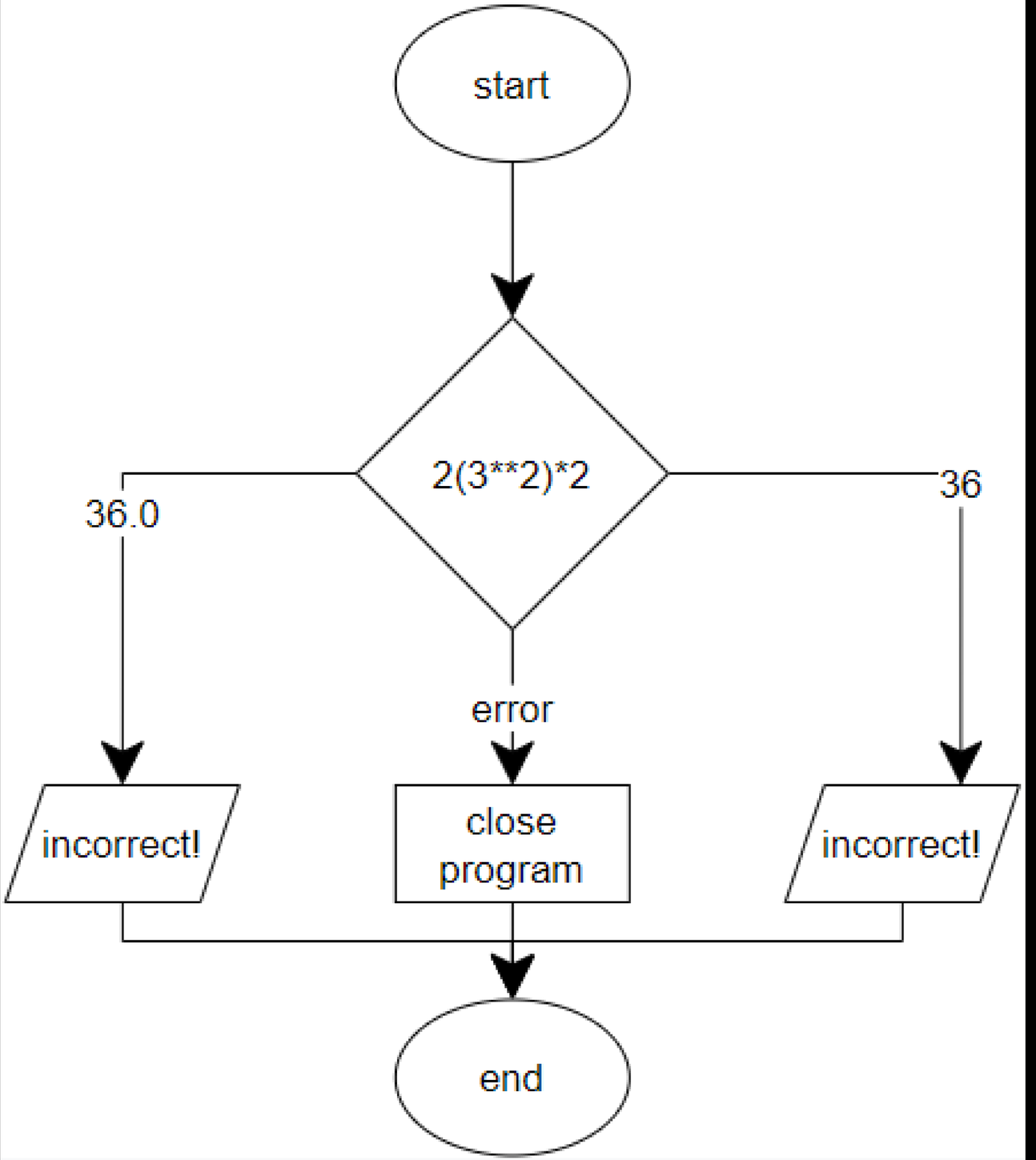
1.1 Conditions



1.1 Conditions



1.1 Conditions



Boolean Algebra

- Introduced by the mathematician George Boole in the book "The Mathematical Analysis of Logic," and "An Investigation of the Laws of Thought"

Boolean Algebra

- Boolean algebra's primary use has been in computer programming languages
- True or False
- 1 or 0
- deals with relational and logical operations

2.2 Recap on Prescedence

Precedence	Operator Sign	Operator Name
Highest	**	Exponentiation
	+x, -x, ~x	Unary positive, unary negative, bitwise negation
	*, /, //, %	Multiplication, division, floor, division, modulus
	+, -	Addition, subtraction
	<<, >>	Left-shift, right-shift
	&	Bitwise AND
	^	Bitwise XOR
	|	Bitwise OR
	==, !=, <, <=, >, >=, is, is not	Comparison, identity
	not	Boolean NOT
	and	Boolean AND
Lowest	or	Boolean OR

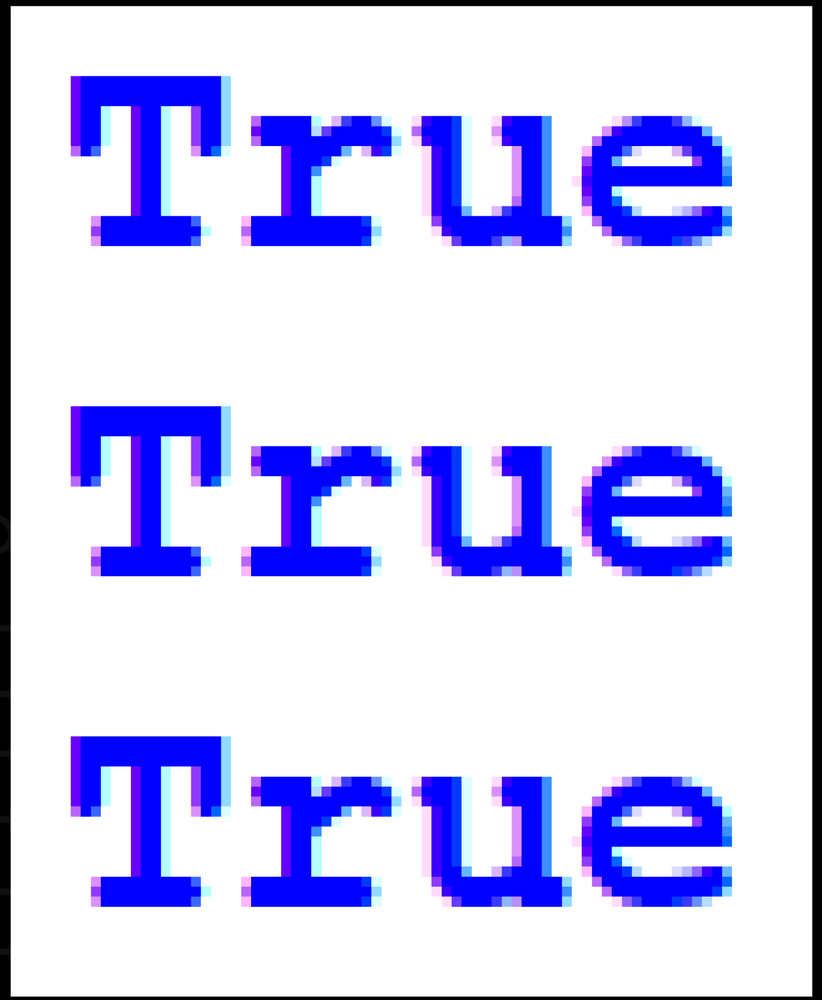
Boolean Algebra

- Usually, relational operators must come first before the logical operators
- Although bitwise logical operators defy this, we work around it via brackets

Boolean Algebra

```
print(8==8)  # True
print(9>=9)  # True
print(8==8 and 9>=9) # True
```

Boolean Algebra



- relational operators work to check variables, logical operators work to check truth values

Boolean Algebra

```
print(True == 'True')
print(True != 'True')
print(1 > 11)
print(11 >= 12)
print(2**3/4*2 == 1.0)
print(1 == True and 0 == False)
print(1 == True ^ 0 == False)
```

2.2 Recap on Prescedence

Precedence	Operator Sign	Operator Name
Highest	**	Exponentiation
	+x, -x, ~x	Unary positive, unary negative, bitwise negation
	*, /, //, %	Multiplication, division, floor, division, modulus
	+, -	Addition, subtraction
	<<, >>	Left-shift, right-shift
	&	Bitwise AND
	^	Bitwise XOR
	|	Bitwise OR
	==, !=, <, <=, >, >=, is, is not	Comparison, identity
	not	Boolean NOT
	and	Boolean AND
Lowest	or	Boolean OR

2.2 Recap on Prescedence

- Take note of the bitwise logical operations

```
False
True
False
False
False
True
False
```

Boolean Algebra

- the main operations are conjunction, disjunction, and negation

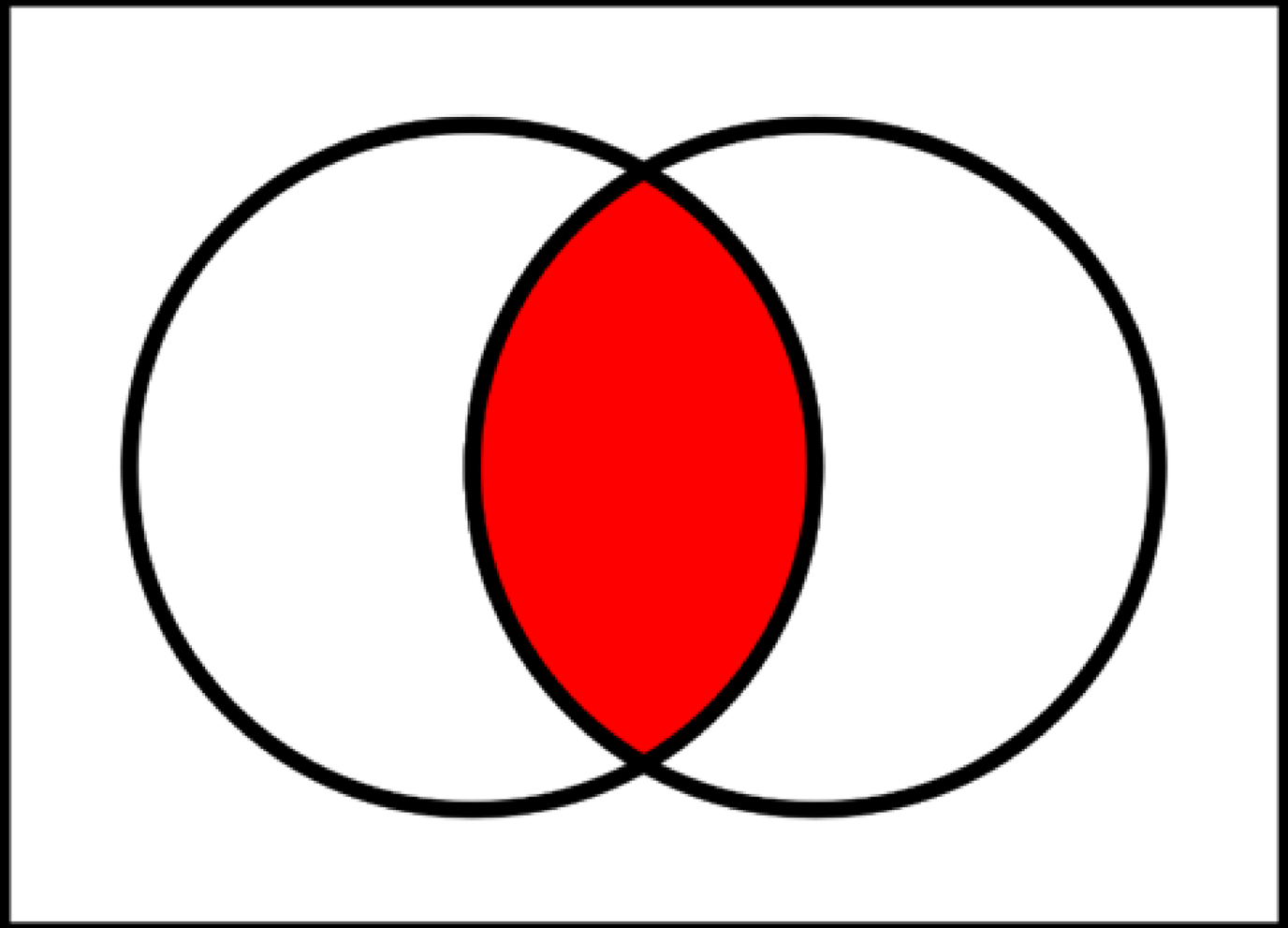
Conjunction

- Logical conjunction is an operation on two logical values, typically the values of two propositions, that produces a value of true if and only if both of its operands are true.

Conjunction

- “True and True to be equal to True”
- may use relational operators, bitwise ampersand (&), or the keyword ‘and’

Conjunction



Conjunction

```
1 print(True and True)
2 print(False and True)
3 print(True and False)
4 print(False and False)
```

Conjunction

```
True

False

False

False
```

Conjunction

Logical 'and'				
L \ R	True	False	n	0
True	True	False	n	0
False	False	False	False	False
n	True	False	n_R	0
0	0	0	0	0

2 Types

- Inclusive and Exclusive

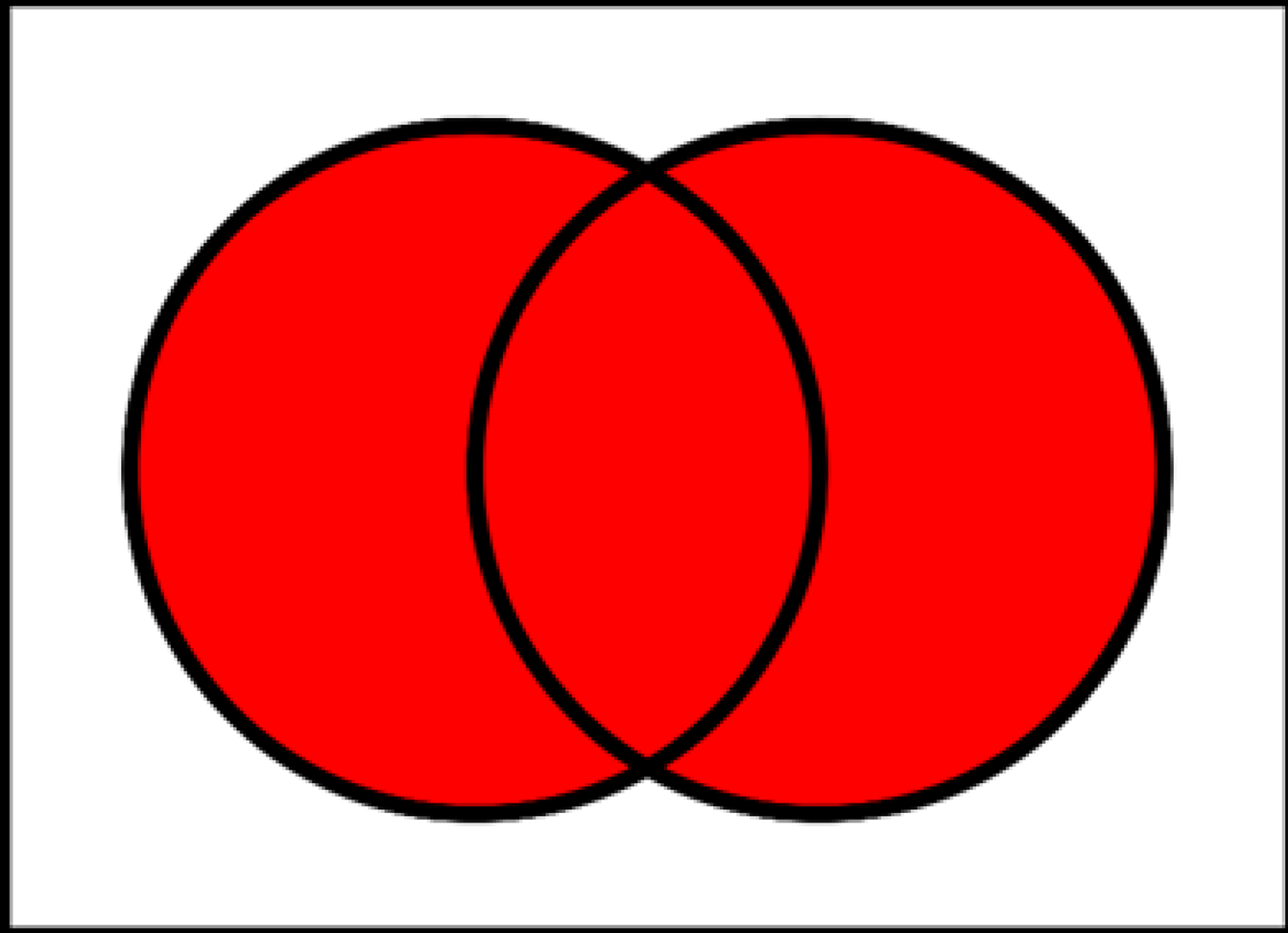
Inclusive

- A disjunction that remains true if either or both of its arguments are true
- Will still be true as long as at least one of the variables are True

Inclusive

- Will only be False if both values are False
- may use relational operators, bitwise vertical bar(|), or the keyword ‘or’

Inclusive



2.4 Disjunction

Inclusive

```
print(1==True or 1==1)
print(0!=True | 1==2)
print(1==True or 1==1 | 0!=True)
```

2.4 Disjunction

Inclusive

```
True
False
True
```

2.4 Disjunction

Inclusive

Logical ‘or’				
L \ R	True	False	n	0
True	True	True	True	True
False	True	False	n	0
n	n	n	n_L	n
0	True	False	n	0

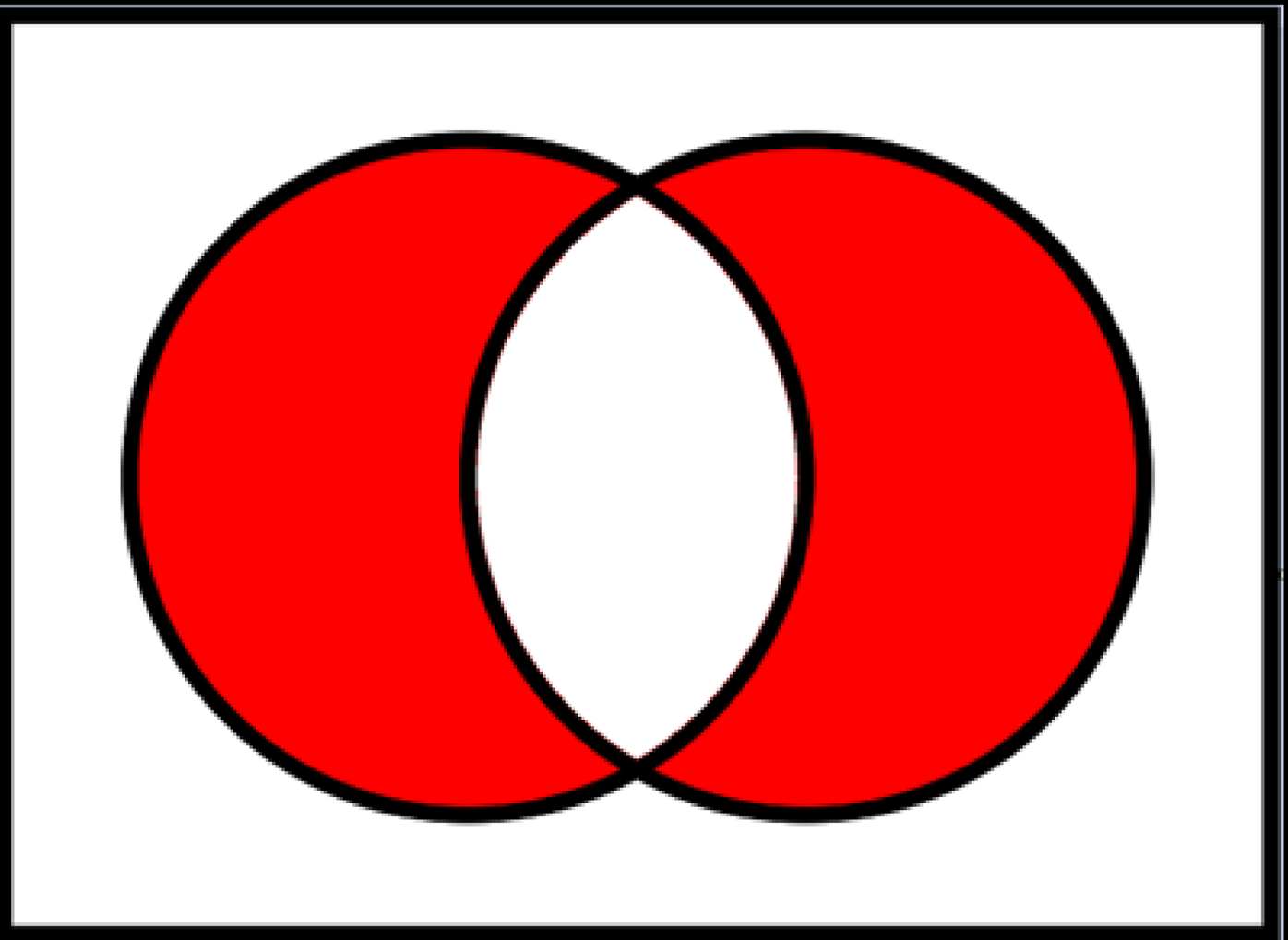
Exclusive

- A disjunction that is true if only one, but not both, of its arguments are true, and is false if neither or both are true, which is equivalent to the XOR connective

Exclusive

- Will be true if only one variable is true
- Will be false if both variables are True or if both variables are False
- may use relational operators, bitwise caret(^) only

Exclusive



2.4 Disjunction

Exclusive

```
print(1==True ^ True==4)
print((1==True) ^ (True==4))
print(5>=5 ^ 6<7 and 3)
print(5>=5 ^ 6<7 or 8>1 and 3)
```

Exclusive

```
False

True

3

True
```

Exclusive

Logical ‘xor’				
L \ R	True	False	n	0
True	False	True	(n-1)	|(n-1)|
False	True	False	n	0
n	(n-1)	n	(n-n)	n
0	1	0	n	0

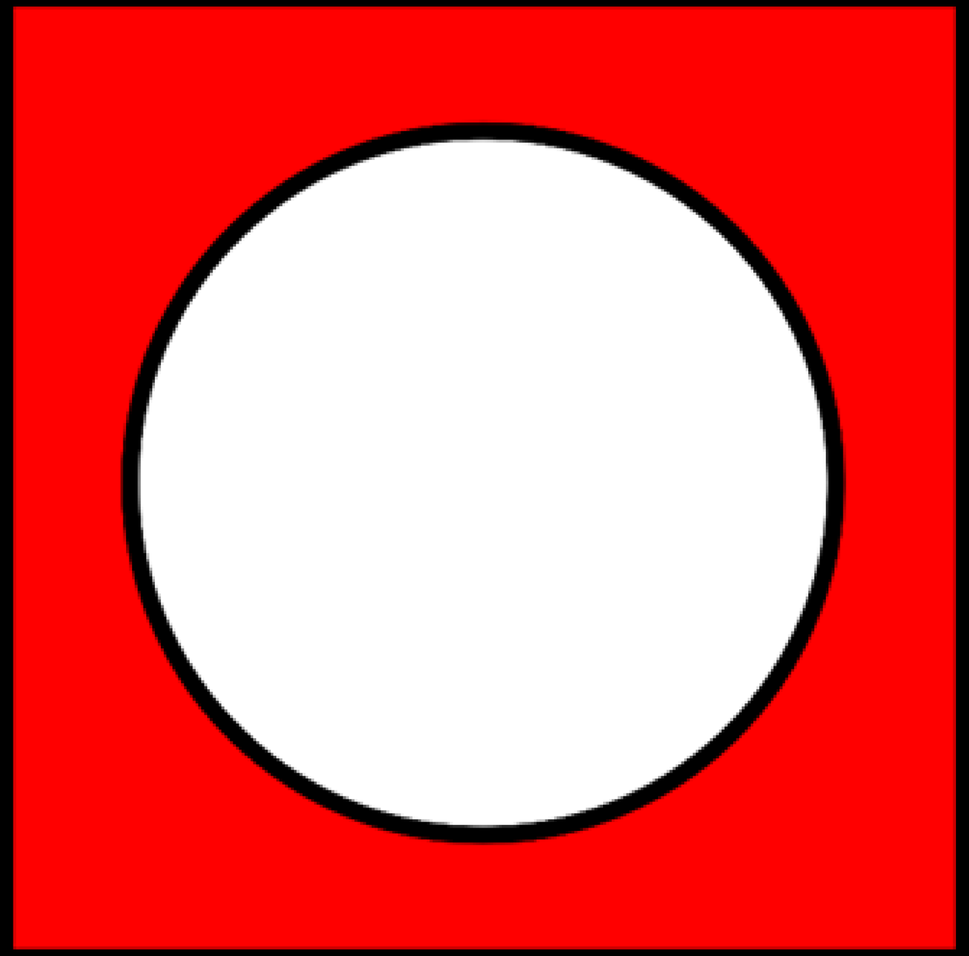
Negation

- Logical negation is an operation on one logical value that produces a value of true when its operand is false and a value of false when its operand is true

Negation

- flips True to False and False to True
- uses the keyword ‘not’

Negation



Negation

```
print(not 1==True ^ True==4)
print(not((1==True) ^ (True==4)))
print(5>=5 ^ 6<7 and not 3)
print(not 5>=5 ^ 6<7 or 8>1 and 3)
```

Negation

```
True
False
False
3
```

Negation

Logical 'not'	
L \ R	not
True	False
False	True
n	False
0	True

3.1 Definition

Conditions

- Conditional expressions, involving keywords such as IF, ELIF, and ELSE, provide Python programs with the ability to perform different actions depending on a Boolean condition

‘if’ Statement

- is one of the most used conditional statements in programming languages
- checks for a given condition, if the condition is True, then the set of code present inside the IF block will

‘if’ Statement

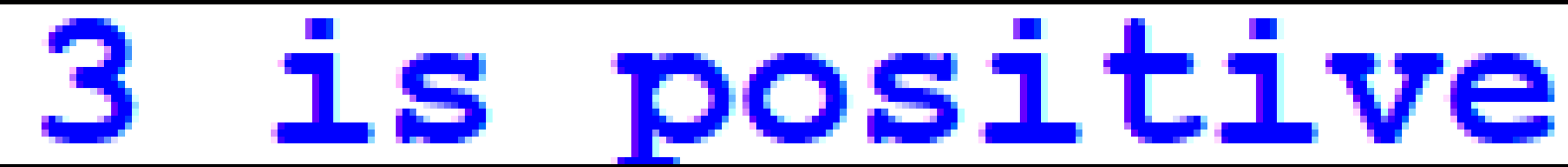
- can be consecutive
- syntax:

```
if(<condition/s>):
    <content>
```

‘if’ Statement

```
1 x = 3
2
3 if x > 0:
4     print(x,'is positive')
```

‘if’ Statement



‘if’ Statement

```
1  x = 3
2
3  if x > 0:
4      print(x, 'is positive')
5  if x > 1:
6      print(x, 'is greater than 1')
7  if x > 2:
8      print(x, 'is greater than 2')
9  if x > 3:
10     print(x, 'is greater than 3')
```

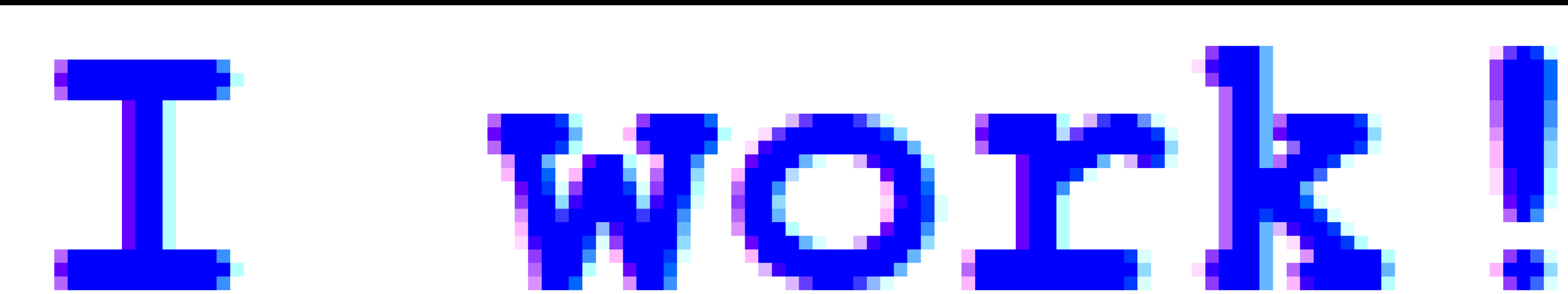
‘if’ Statement

```
3 is positive
3 is greater than 1
3 is greater than 2
```

‘if’ Statement

```
1 x = 3
2
3 if x!=True ^ True!=3:
4     print('I work!')
```

‘if’ Statement



‘if … else’ Statement

- is used if none of the previous conditions are met
- is a ‘fail-safe’, will execute if conditions above are all False
- is optional in a ‘if-else’ code block
- 'else' Statement is optional

‘if … else’ Statement

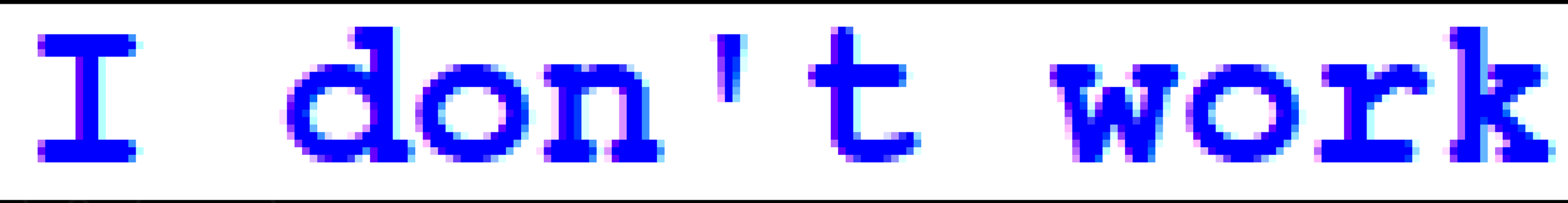
- syntax:

```
if(False):
        <content that will not run>
else:
        <content that will run>
```

‘if … else’ Statement

```
x = 3


if x!=True ^ True==3:
    print('I work!')
else:
    print("I don't work")
```

‘if … else’ Statement



‘if … elif’ Statement

```
1 x = 3
2
3 if x > 0:
4     print(x, 'is positive')
5 if x > 3:
6     print(x, 'is greater than 3!')
7 else:
8     print(x, 'is less than 4!')
```

‘if … elif’ Statement

```
3 is positive
3 is less than 4!
```

‘if … elif’ Statement

- is used to check additional conditions if the first condition is evaluated as False
- ‘else if’
- is used only under an ‘if’ statement and can have multiple ‘elif’ Statements

‘if … elif’ Statement

- syntax:

```
if(False):
        <content that will not be executed>
elif(<additional condition that might be True>):
        <content that might be executed>
```

‘if … elif’ Statement

```
1 x = 3
2
3 if x>0:
4     print('this is positive')
5 elif x==3:
6     print('this is 3')
7 else:
8     print('this is not a number')
```

‘if … elif’ Statement

```
this is positive
```

‘if…elif…else’ Statement

- note that there can only be one ‘if’ statement, one or multiple optional ‘elif’ statements, and an optional ‘else’ statement

‘if…elif…else’ Statement

- syntax:

```
if(<condition>):
        <content that might run>
<elif(<other condition>):
        <content that might run>
>
<else:
        <content that might run>
>
```

‘if…elif…else’ Statement

```
1 x = 3
2
3 if x > 0:
4     print(x, 'is positive')
5 elif x > 3:
6     print(x, 'is greater than 3!')
7 else:
8     print(x, 'is less than 4!')
```

‘if…elif…else’ Statement

```
3 is positive
```

‘Match-Case’ Statement

- from version 3.10 upwards, Python has implemented a switch case feature called “structural pattern matching”.
- it works much like consecutive ‘if’ statements, will check one-by-one all ‘case’s even if a case has already been True

‘Match-Case’ Statement

- it tries to see(match) if a variable ‘is equal to’ the following cases

‘Match-Case’ Statement

- the optional underscore(_) symbol is used to denote “default”
- imagine it like the ‘else’ statement

‘Match-Case’ Statement

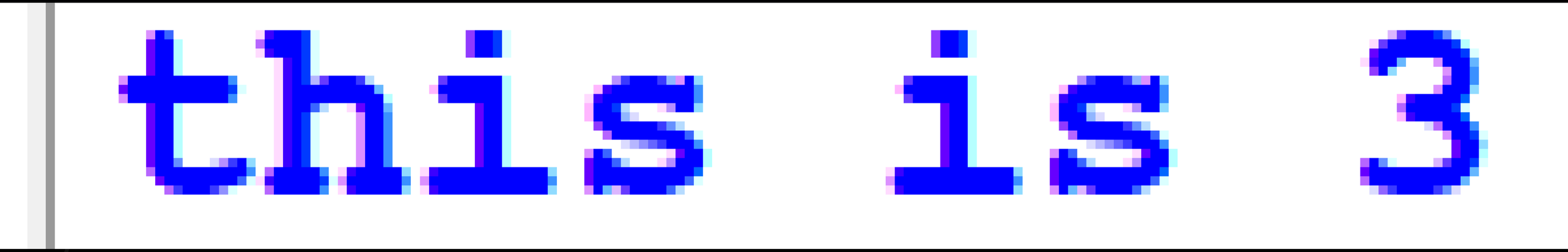
- syntax:

```
match <variable_name>:
        case <condition>:
                <content>
                ...
        case _:
                <content>
```

‘Match-Case’ Statement

```
1 x = '3'
2
3 match x:
4     case '3':
5         print('this is 3')
6     case _:
7         print('this is not 3')
```

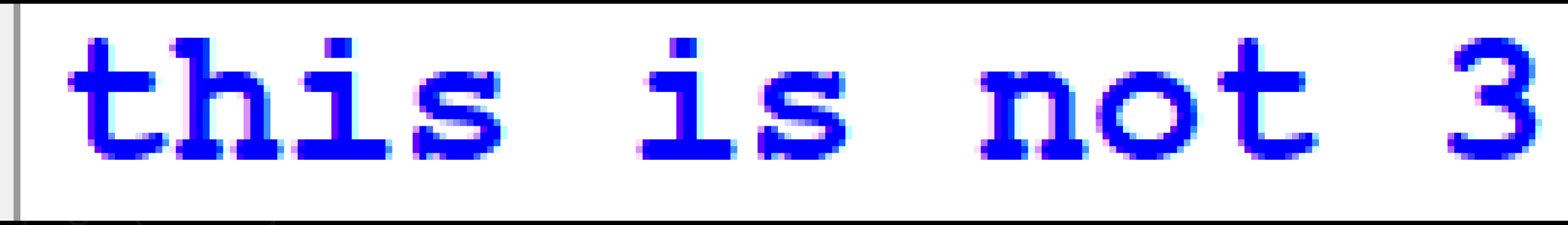
‘Match-Case’ Statement



‘Match-Case’ Statement

```
1 x = 3
2
3 match x:
4     case '3':
5         print('this is 3')
6     case _:
7         print('this is not 3')
```

‘Match-Case’ Statement



Nesting

- code block structures like ‘try-except’ and ‘if’ statements create indentations below them to denote that the following lines are part of the code block

Nesting

- it is where code blocks fit in other code blocks, structures inside other structures

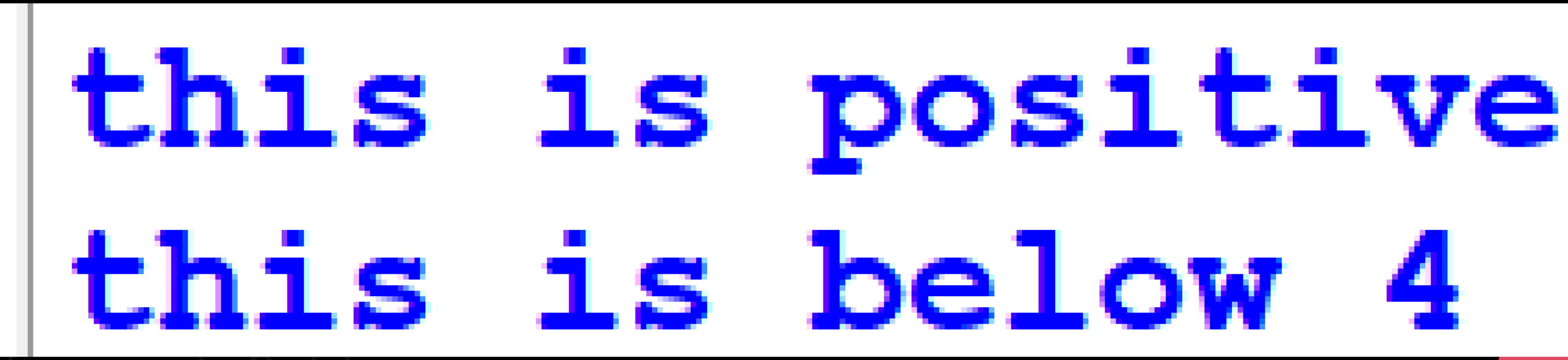
Nesting

- if that is the case, can we put ‘if’ statements inside another ‘if’ statement? A ‘match-case’ inside an ‘if’ statement? An ‘if’ statement inside a ‘match-case’?

Nesting

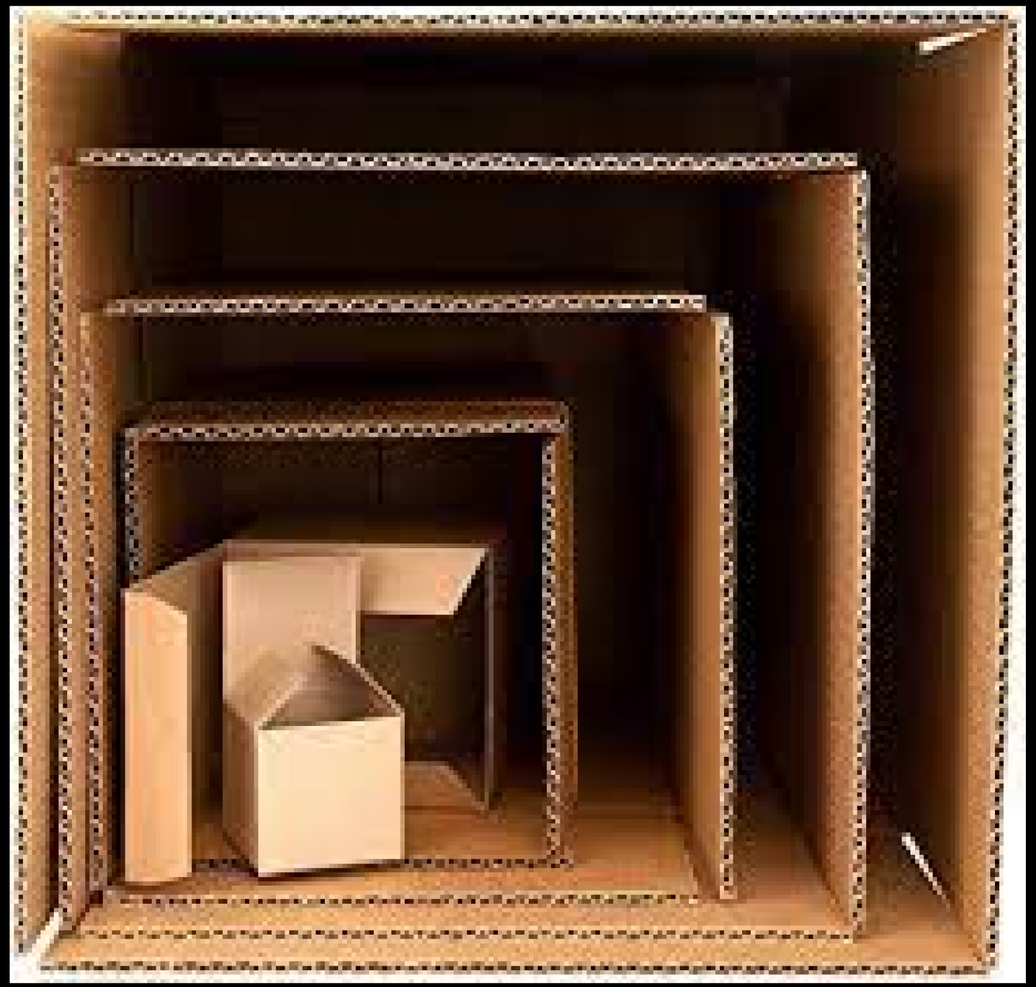
```
x = 3

if x > 0:
    print('this is positive')
    if x < 4:
        print('this is below 4')
    else:
        print("i don't know, sir")
else:
    print('Happy Birthday!')
```

Nesting



Nesting

Nesting



REPETITION

PREPARED BY: LUIS MEING

OBJECTIVES:

01 **Definition**

02 **Parts of a Loop**

03 **Types of Loops**

04 **Repetition Structures in Python**

05 **Loop Nesting**

1.1 Definition

Repetition

- the recurrence (duplication) of an action or event

1.1 Definition

Repetition Control Structures

- repetition control structures are also referred to as iterative structures

- these are groups of code which are designed to repeat a set of statements

1.1 Definition

Repetition Control Structures

- this repetition can repeat zero or more times, until some control value or condition causes the repetition to cease

1.1 Definition

Loops

- most common term to describe repetition control structures

Declaration of ‘counter’ variable

- the creation of a variable that will be compared every time the body is executed at least once

Condition

- the logic that evaluates True or False
- if the condition is True, the loop will repeat
- if the condition is False, the loop will end and exit the structure

Body

- the code block/s within the loop

‘counter’ Variable Update

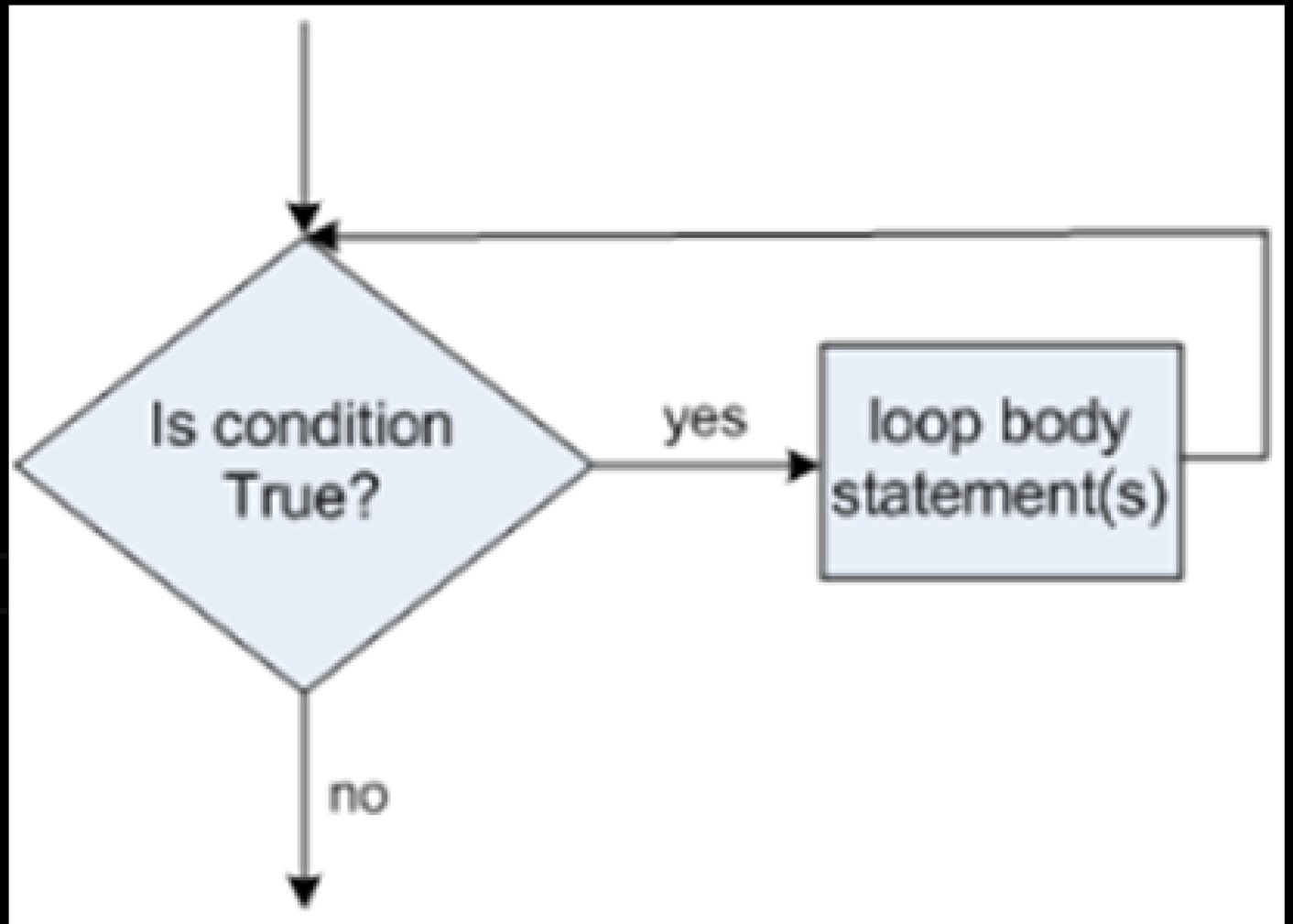
- the change that happens to the ‘count’ variable if the condition is true

- can use arithmetic operators, direct assignment via a conditional structure, or special keywords

Pre-test

- can execute its body a minimum of zero times, if the initial condition evaluates to False

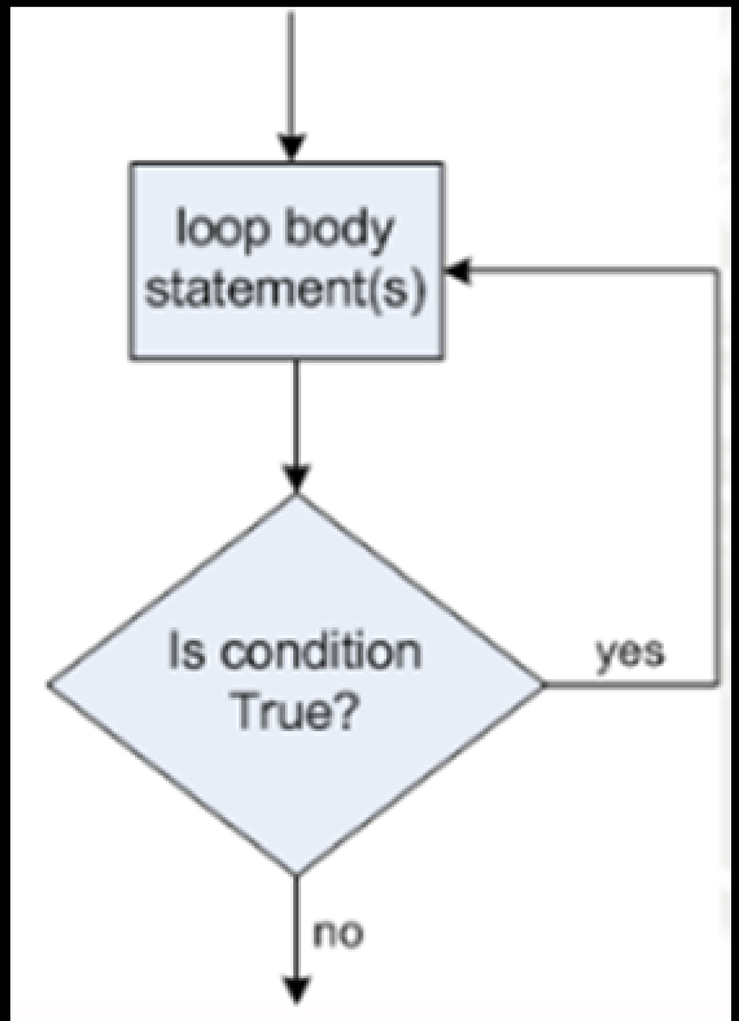
Pre-test



Post-test

- can execute its body a minimum of one time even if the initial condition evaluates to False

Post-test



Post-test

- Python does not have a Post-test loop but an example is Java’s ‘do-while’ Loop

‘while’ Loops

- is a statement in Python that repeatedly executes a block of statements if a test at the top keeps being True

‘while’ Loops

- when the test in the ‘while’ condition becomes False, the process continues to the statement that is below the ‘while’ block

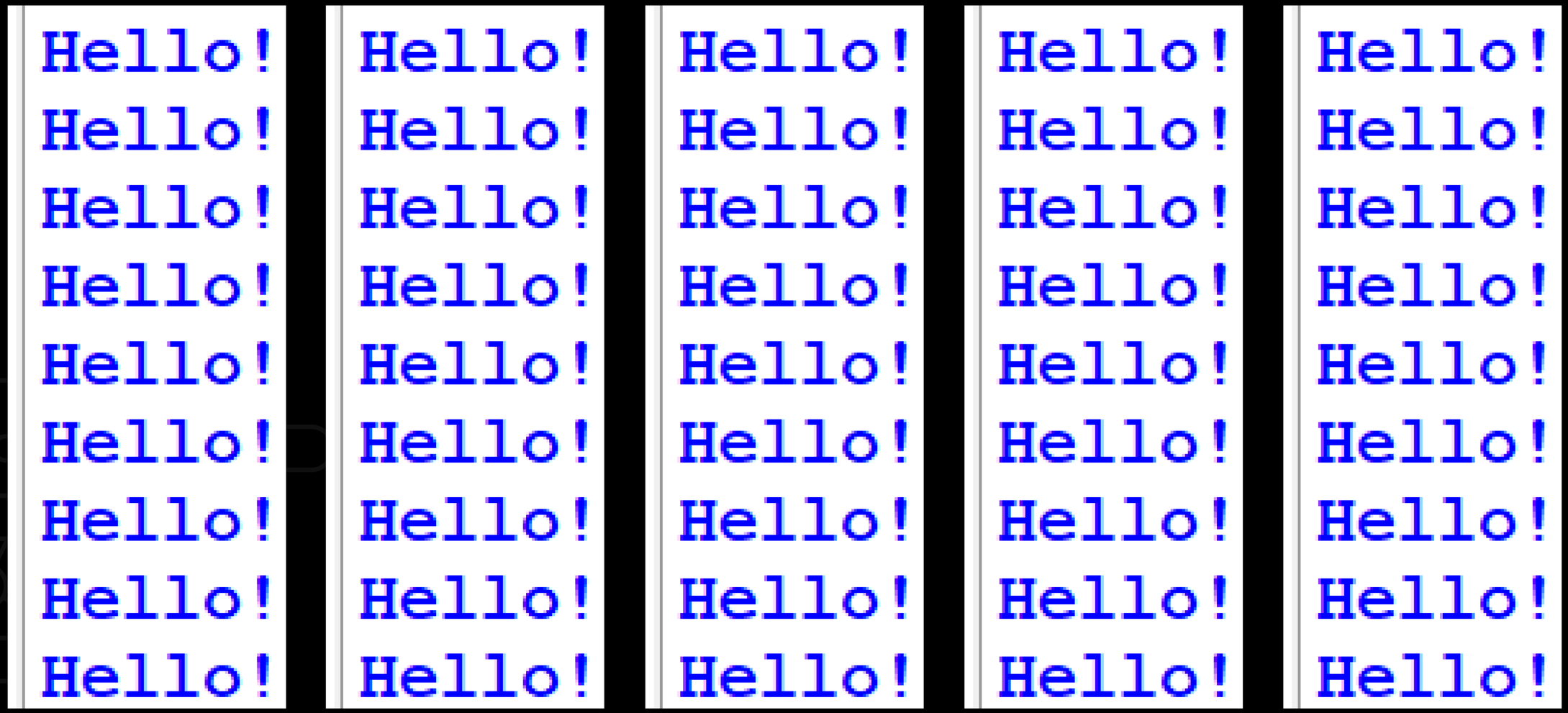
‘while’ Loops syntax

<‘counter’ Variable Update>
while(<condition>)
    <body>
    <‘counter’ Variable Update >

‘while’ Loops syntax

```
1 i = 0
2
3 while i < 5:
4         print("Hello!")
```

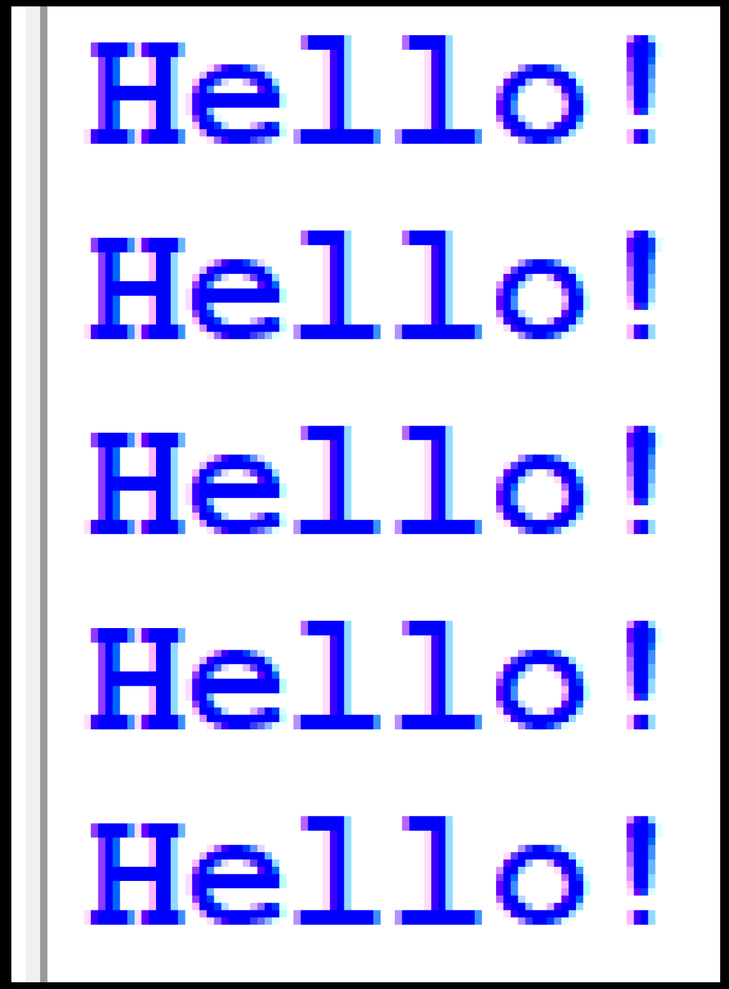
‘while’ Loops syntax



‘while’ Loops syntax

```
1 i = 0
2
3 while i < 5:
4         print("Hello!")
5         i += 1
```

‘while’ Loops syntax



‘while’ Loops syntax

```
1  my_list = [1, 2, 3, 4, 5]
2
3  i=0
4
5  while i < len(my_list):
6      if my_list[i]%2==0:
7          print(my_list[i], "is even")
8      else:
9          print(my_list[i], "is not even")
10     i=i+1
```

‘while’ Loops syntax

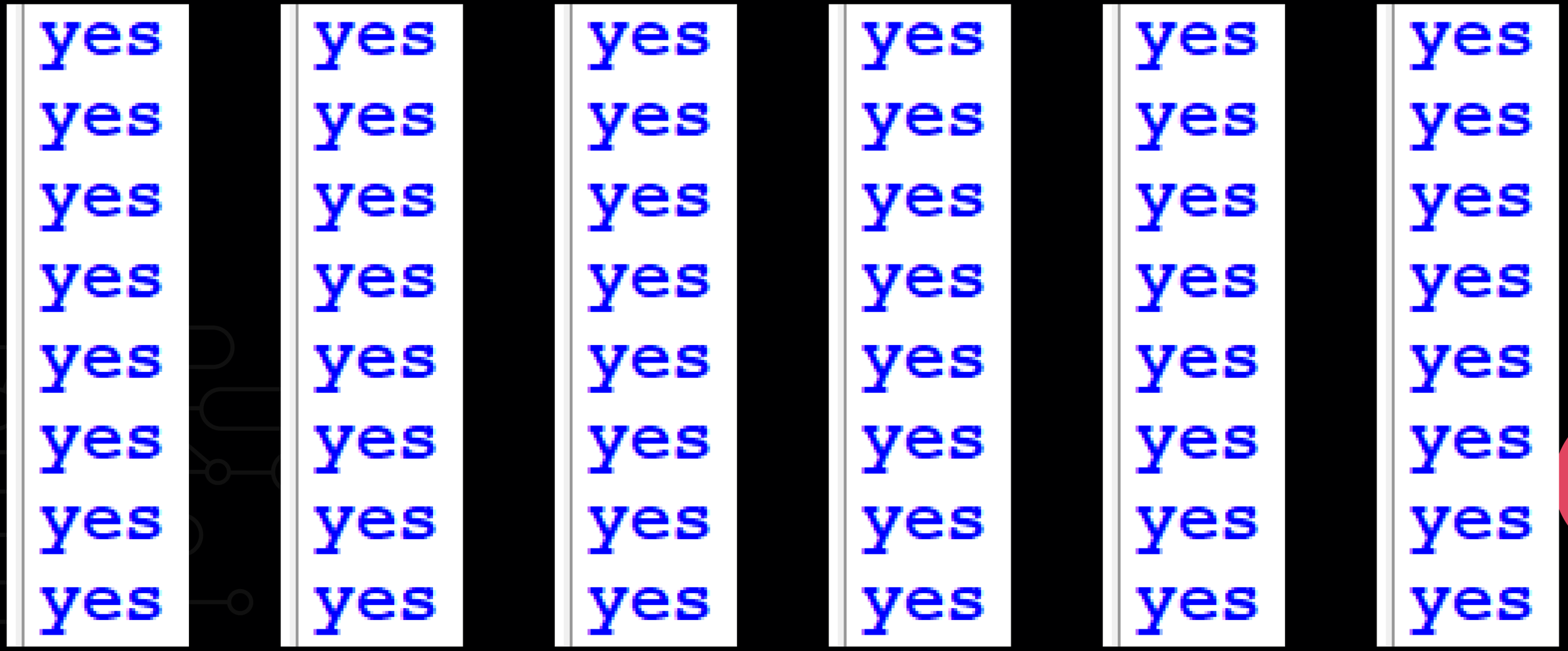
Or

while(<condition>)
    <body>

‘while’ Loops syntax

```
1 while True:
2     print("yes")
```

‘while’ Loops syntax



‘while’ Loops Optional Keywords

- ‘break’ Keyword
  - stops the loop, exits the structure
  - used only inside the loop

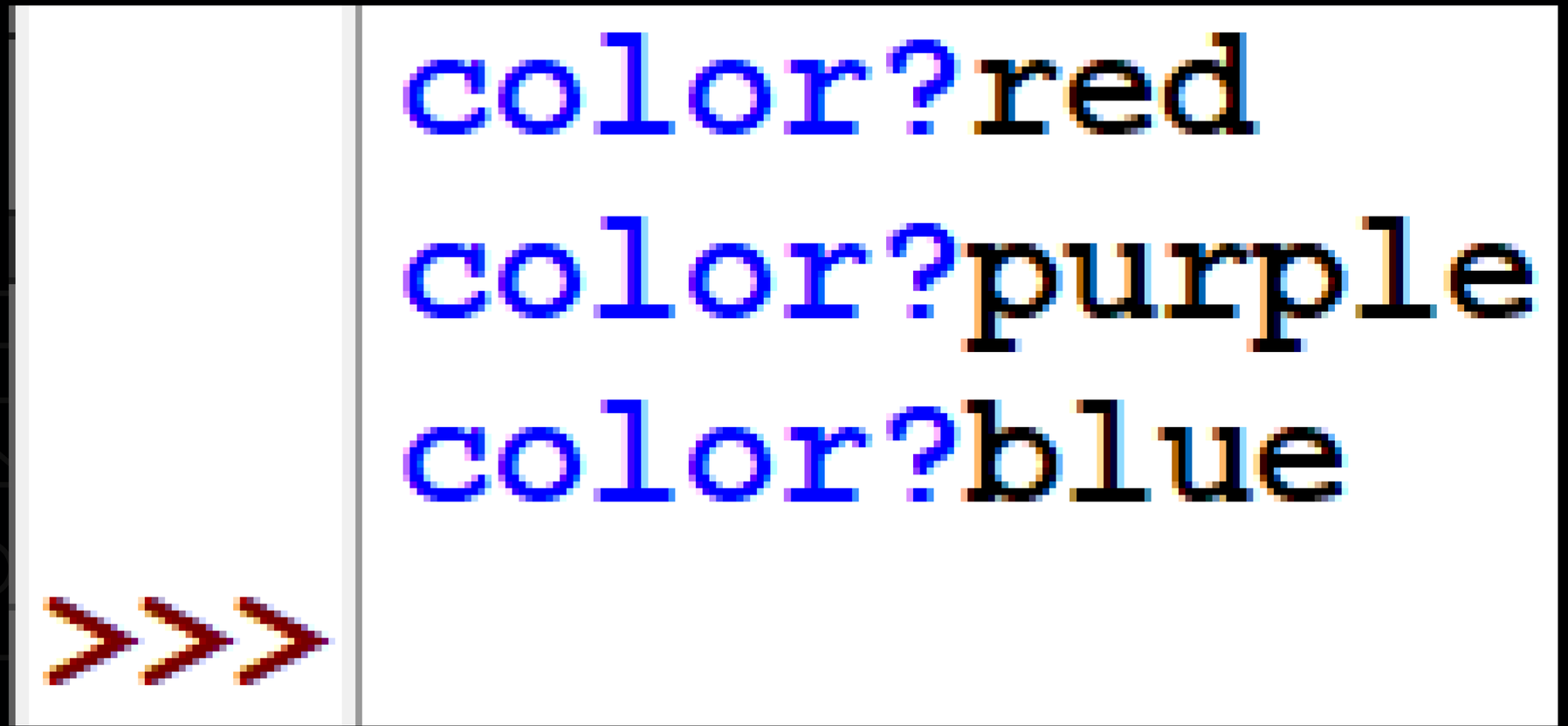
‘while’ Loops Optional Keywords

- ‘break’ Keyword

```
1 while True:
2         response=input("color?")
3         if response=="blue":
4                 break
```

‘while’ Loops Optional Keywords

- ‘break’ Keyword



‘while’ Loops Optional Keywords

- ‘continue’ Keyword
  - goes directly to the top, checking the condition and will execute code again if condition is False
  - used only inside the loop

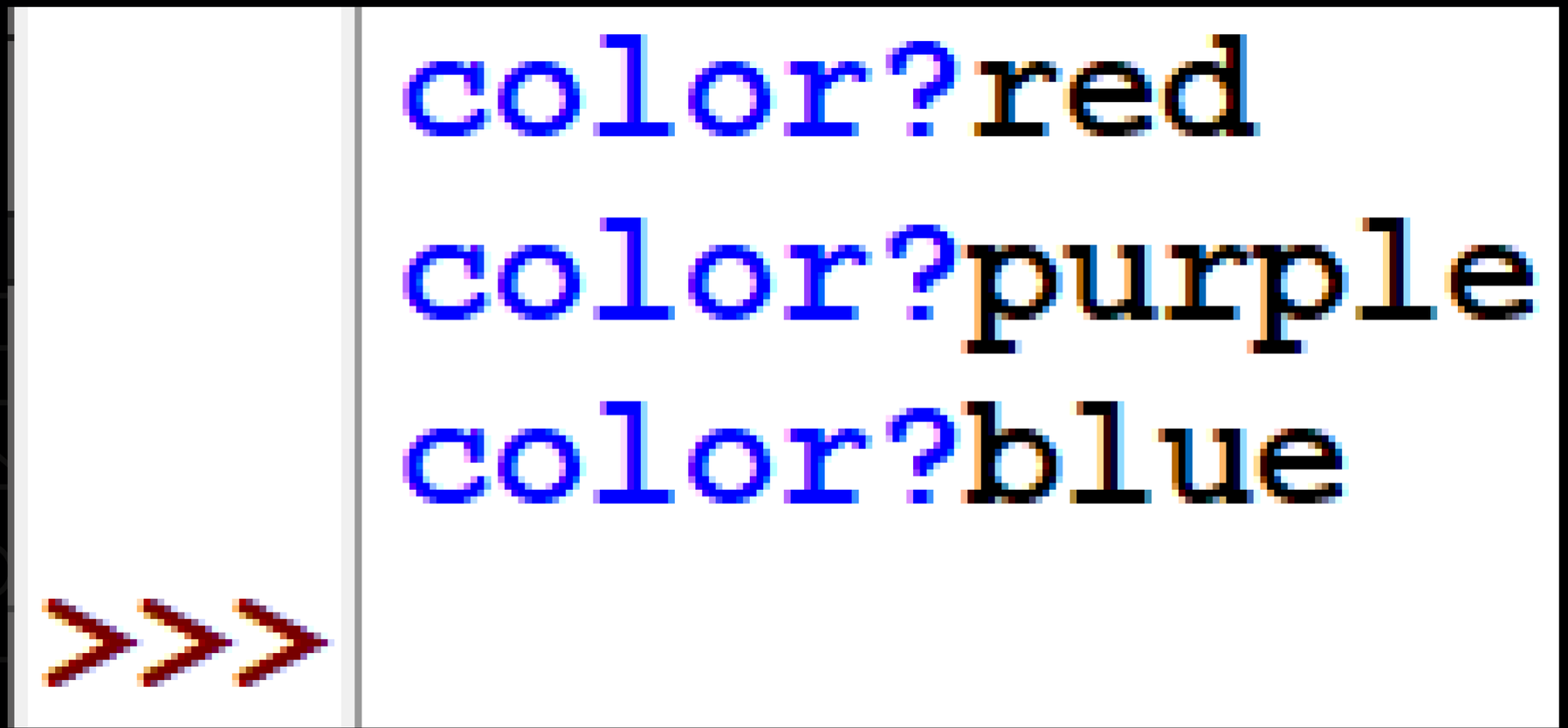
‘while’ Loops Optional Keywords

- ‘continue’ Keyword

```
1 while True:
2     response=input("color?")
3     if response=="blue":
4         break
5     else:
6         continue
7     print("I am ignored")
```

‘while’ Loops Optional Keywords

- ‘continue’ Keyword



‘while’ Loops Optional Keywords

- ‘else’ Keyword
  - this runs if and only if the loop is exited, exited by not triggering a ‘break’ Keyword
  - is used outside the loop indentation

‘while’ Loops Optional Keywords

- ‘else’ Keyword

```
1 i = 0
2 while i<3:
3     response=input("color?")
4     if response=="blue":
5         break
6     i += 1
7 else:
8     print("successful 3 iterations")
```

‘while’ Loops Optional Keywords

- ‘else’ Keyword

```
color?red
color?purple
color?green
successful 3 iterations
```

```
>>> color?blue
```

‘while’ Loops Optional Keywords

- ‘pass’ Keyword
  - literally, does nothing but is used as a substitute to occupy code blocks that require content

‘while’ Loops Optional Keywords

- ‘pass’ Keyword

```
1 i = 0
2 while i<3:
3     response=input("color?")
4     if response=="blue":
5         pass
6     else:
7         print("this is not blue!")
8     i += 1
```

‘while’ Loops Optional Keywords

- ‘pass’ Keyword

```
color?red
this is not blue!
color?green
this is not blue!
color?purple
this is not blue!
>>>
```

```
color?blue
color?blue
color?blue
>>>
```

‘for’ Loops

- usually used to go through sequence structures
- also uses optional keywords ‘break’, ‘continue’, ‘else’, ‘pass’
- automatically increments the counter variable

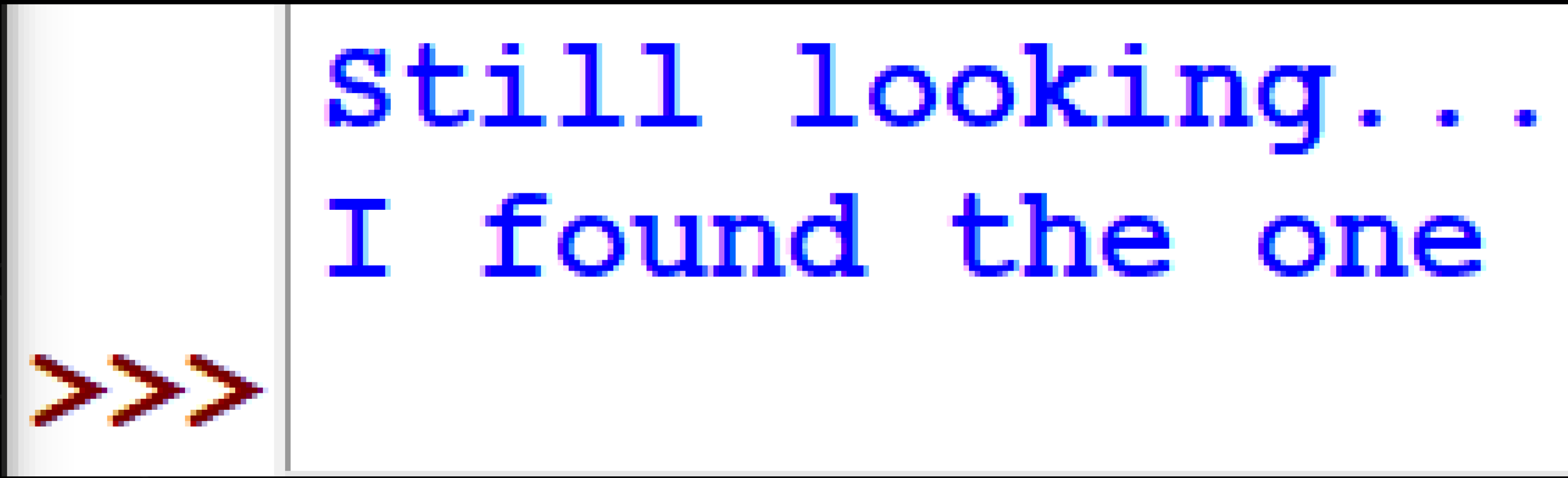
‘for’ Loops syntax

```
for <any_other_var_name> in <structure>:
        <content>
```

‘for’ Loops syntax

```
1 my_list = [1, 2, 3, 4, 5]
2
3 for i in my_list:
4     if i == 2:
5         print("I found the one")
6         break
7     else:
8         print("Still looking...")
```

‘for’ Loops syntax



Loop Nesting

- can use ‘for’ and ‘while’ Loops
- much like nested ‘if’ statements, there can be loops inside other loops

Loop Nesting

- there can be a ‘while’ Loop inside a ‘while’ Loop, a ‘for’ Loop inside a ‘for’ Loop, a ‘for’ Loop inside a ‘while’ Loop, a ‘while’ Loop inside a ‘for’ Loop and so on and so forth.
- take note of indentations, they make a difference between consecutive loops instead of nested loops and vice versa.

Loop Nesting

```
1  my_list = [1, 2, 3, 4, 5]
2
3  i=0
4
5  while i < len(my_list):
6      if my_list[i]%2==0:
7          print(my_list[i], "is even")
8      else:
9          print(my_list[i], "is not even")
10     for x in my_list:
11         print(my_list[i])
12     i=i+1
```

FUNCTION

PREPARED BY: LUIS MEING

OBJECTIVES:

01 Definition

02 Built-in Functions

03 User-defined Functions

04 Nested Functions

05 Variable Scope

06 Compare and Contrast

1.1 Definition

Functions

- are 'groups of code blocks' with a name

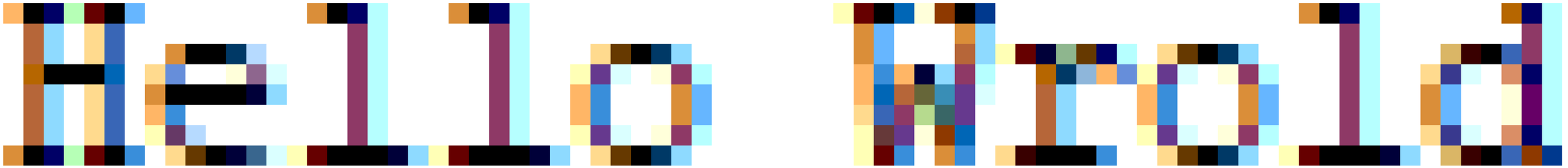
1.1 Definition

Functions

```
1  class Demo{
2      public static void main(String args[]){
3          System.out.println("Hello Wrold");
4      }
5  }
```

1.1 Definition

Functions



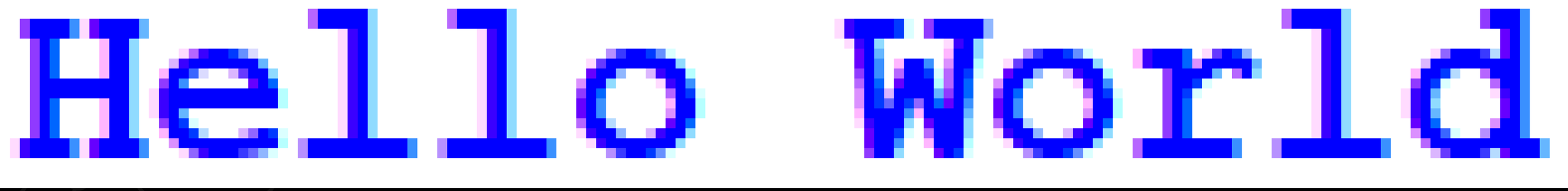
1.1 Definition

Functions

```
print("Hello World")
```

1.1 Definition

Functions



Functions vs. Methods

- How many are the functions?

- How many are the methods?

1.1 Definition

Functions vs. Methods

- A function is a set of instructions or procedures to perform a specific task

- A method is a set of instructions that are associated with an object.

Built-in Functions

- functions that are part of the 'libraries' you downloaded when you started to program in that language

Built-in Functions

```
System.out.println("Hello Wrold");
```

```
print("Hello World")
```

User-defined Functions

- take note to not use keywords to name your user-defined functions

User-defined Functions Syntax

- In Python, create as:

```
def <function_name>(<parameters>):
        <content>
```

User-defined Functions Syntax

```
def hello_world():
    print("Hello World")
```

User-defined Functions Syntax

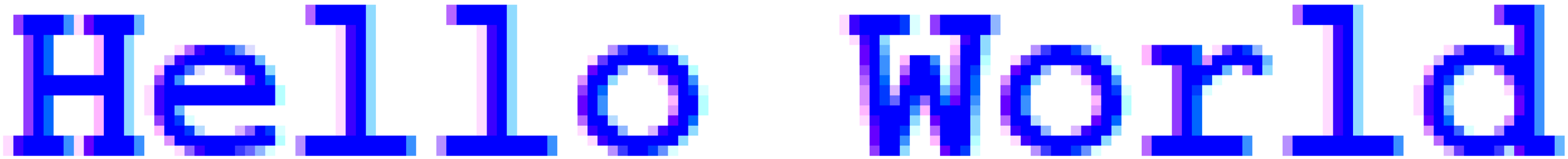
- In Python, refer as :

<function_name>(<arguments>)

User-defined Functions Syntax

```
def hello_world():
    print("Hello World")


hello_world()
```

User-defined Functions Syntax



Arguments vs. Parameters

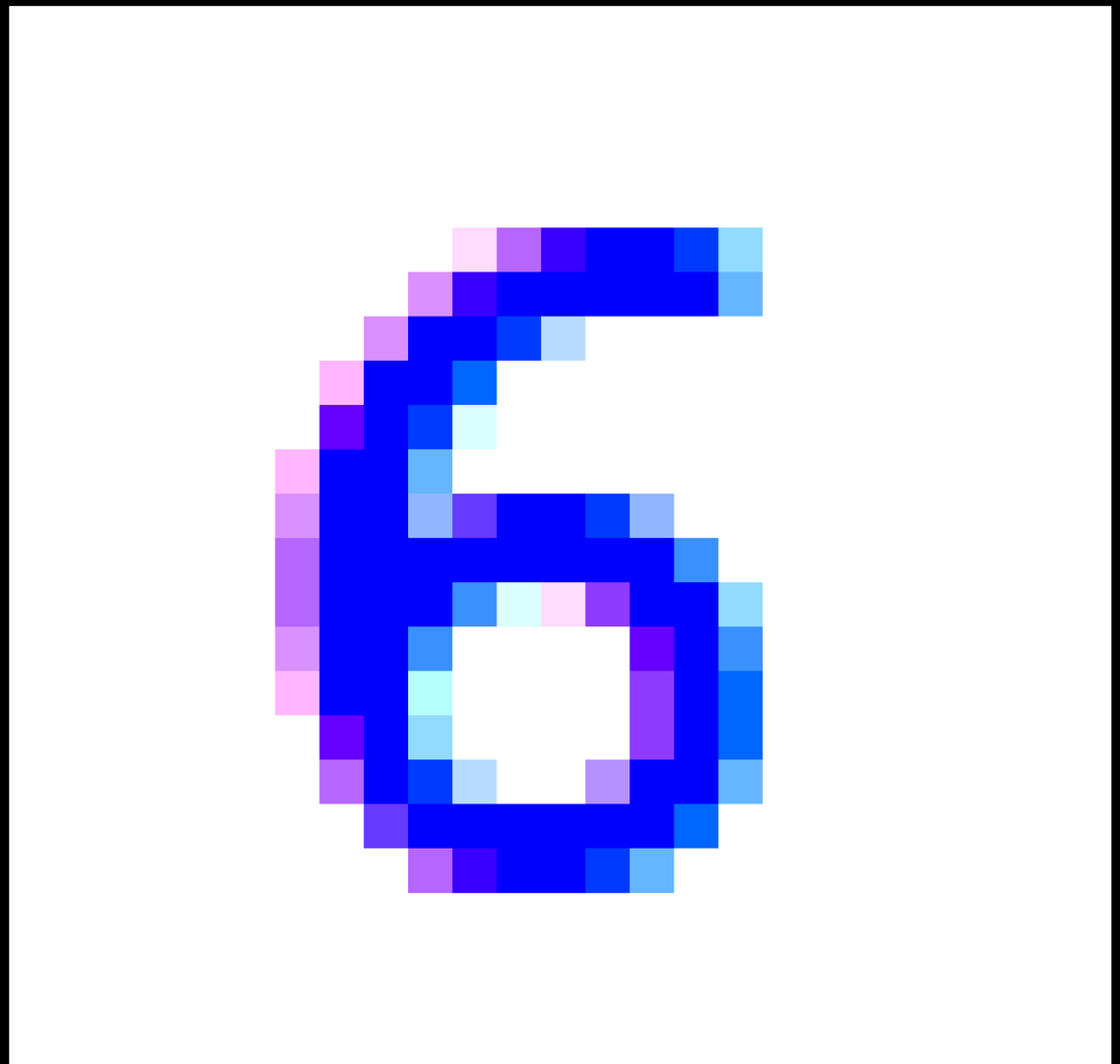
- arguments are values put inside the parentheses when referring to a function

- parameters are the variables put inside the parentheses when creating a function

Arguments vs. Parameters

```
1 def multiply(x, y):
2     return x*y
3
4 a = 2
5 b = 3
6 print( multiply(a, b) )
```

Arguments vs. Parameters



Return

- refers to what the function will give back as a value when the function is used
- is optional, uses the ‘return’ Keyword

- unlike other languages, you do not need to explicitly declare a return type of the function in Python

Functions with an Arbitrary Number of Arguments

- sometimes, there may not be argument that will be passed to the function
- in Python, we use the '*' symbol as a prefix

Functions with an Arbitrary Number of Arguments

```
1 def multi_args(*args):
2     for i in args:
3         print(i)
```

Functions with an Arbitrary Number of Arguments

- we can pass primitives and non-primitive values

Functions with an Arbitrary Number of Arguments

```
1 def multi_args(*args):
2     for i in args:
3         print(i)
4
5
6 sample_list = [5, 8, 10, 12, 15]
7 multi_args(sample_list)
8 multi_args(*sample_list)
```

Functions with an Arbitrary Number of Arguments

```
[5, 8, 10, 12, 15]
5
8
10
12
15
```

Functions with an Arbitrary Number of Arguments

- when passing mutable structures, take note:

- Mutating a parameter will mutate the argument
- reassigning the parameter won’t reassign the argument

Functions with an Arbitrary Number of Arguments

```
1 def sample(x):
2     x[0] = 10
3     x = [20, 30, 40]
4     x[1] = 15
5     print(x)
6
7 y = [5, 6, 7]
8 sample(y)
9 print(y)
```

Functions with an Arbitrary Number of Arguments

```
[20, 15, 40]
[10, 6, 7]
```

User-defined Functions Syntax

- In Java, create as:

```
<access_modifier> <static> <return_type><method name>([parameters]){
        <method body>
}
```

User-defined Functions Syntax

```
<access_modifier> <static> <return_type><method name>([parameters]){
    <method body>
}
```

- access modifier
  - either public or private
  - by default is public

User-defined Functions Syntax

```
<access_modifier> <static> <return_type><method name>([parameters]){
    <method body>
}
```

- static
  - will determine if the method is 'static' or an 'instance' method

User-defined Functions Syntax

```
<access_modifier> <static> <return_type><method name>([parameters]){
    <method body>
}
```

- return type
  - refers to what the function will give back as a value when called
  - uses the 'return' keyword is uses a data type

User-defined Functions Syntax

```
public static void main(String args[]){
    System.out.println("Hello Wrold");
}
```

User-defined Functions Syntax

```
1   package com.mycompany.demonstration;
2
3   public class Demonstration {
4       static String hello(){
5           return "Hello Wrold";
6       }
7
8       public static void main(String args[]){
9           hello();
10      }
11  }
```

User-defined Functions Syntax

```
------------------------------------------------------------------------

BUILD SUCCESS

------------------------------------------------------------------------

Total time:  3.959 s
Finished at: 2022-11-11T08:03:14+08:00

------------------------------------------------------------------------
```

User-defined Functions Syntax

```
1   package com.mycompany.demonstration;
2
3   public class Demonstration {
4       static String hello(){
5           return "Hello Wrold";
6       }
7
8       public static void main(String args[]){
9           System.out.println(hello());
10      }
11  }
```

User-defined Functions Syntax

```
Building Demonstration 1.0-SNAPSHOT
--------------------------------[ jar ]---------

--- exec-maven-plugin:3.0.0:exec (default-cli)
Hello Wrold
------------------------------------------------
BUILD SUCCESS
------------------------------------------------
Total time:  4.208 s
Finished at: 2022-11-11T08:05:42+08:00
------------------------------------------------
```

User-defined Functions Syntax

```
1   package com.mycompany.demonstration;
2
3   public class Demonstration {
4       static void hello(){
5           System.out.println("Hello Wrold");
6       }
7
8       public static void main(String args[]){
9           hello();
10      }
11  }
```

User-defined Functions Syntax

```
Building Demonstration 1.0-SNAPSHOT
--------------------------------[ jar ]---------

--- exec-maven-plugin:3.0.0:exec (default-cli)
Hello Wrold
------------------------------------------------
BUILD SUCCESS
------------------------------------------------
Total time:  2.189 s
Finished at: 2022-11-11T08:08:17+08:00
------------------------------------------------
```

User-defined Functions Syntax

```
1   package com.mycompany.demonstration;
2
3   public class Demonstration {
4
5       public static void main(String args[]){
6           char hello[] = {'h','e','l','l','o'};
7
8           System.out.println(hello.length);
9           System.out.println(hello);
10      }
11  }
```

User-defined Functions Syntax

```
Building Demonstration 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ Demonstration ---
5
hello
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.487 s
Finished at: 2022-11-11T08:19:36+08:00
------------------------------------------------------------------------
```

Nested Functions

- you can place functions inside another function
- this is possible in all programming languages that follow OOP
- take note that parameters passed to the outer function are also passed to the inner function/s

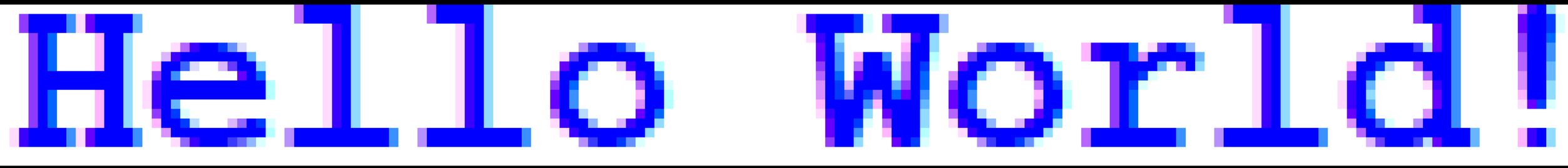
Nested Functions

- you can place functions inside another function
- this is possible in all programming languages that follow OOP
- take note that parameters passed to the outer function are also passed to the inner function/s

4.1 Nested Functions

```
def outer_func(who):
    def inner_func():
        print("Hello", who)
    inner_func()


outer_func("World!")
```

Nested Functions



4.1

```
1  package com.mycompany.demonstration;
2
3  public class Demonstration {
4      static void Foo() {
5          class Local {
6
7              void fun() {
8                  System.out.println("Hello");
9              }
10         }
11         new Local().fun();
12     }
13
14     public static void main(String[] args) {
15         Foo();
16     }
17 }
```

Nested Functions

```
-----------------------< com.mycompany:Demonstration >-----------------------
Building Demonstration 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ Demonstration ---
Hello
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.771 s
Finished at: 2022-11-11T14:05:14+08:00
------------------------------------------------------------------------
```

Variable Scope

- Determines where in the program the variable is accessible
- Determines the lifetime (how long the variable can exist in memory)

- Determined by placement of the variable or where the variable is declared

Variable Scope

- Determines where in the program the variable is accessible
- Determines the lifetime (how long the variable can exist in memory)

- Determined by placement of the variable or where the variable is declared

Variable Scope

```
1   package com.mycompany.demonstration;
2
3   public class Demonstration {
4       public static void main(String args[]){
5           for(int i=0;i<5;i++){
6               System.out.println(i);
7           }
8           System.out.println(i);
9       }
10  }
```

Variable Scope

```
-------------------< com.mycompany:Demonstration >--------------------
Building Demonstration 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ Demonstration ---
Exception in thread "main" java.lang.RuntimeException: Uncompilable code
- cannot find symbol
  symbol:   variable i
  location: class com.mycompany.demonstration.Demonstration
        at com.mycompany.demonstration.Demonstration.main(Demonstration.j
ava:1)
Command execution failed.
```

Variable Scope

```
1   package com.mycompany.demonstration;
2
3   public class Demonstration {
4       public static void main(String args[]){
5           int a=0;
6           if (a==0){
7               int i = 5;
8           }
9           System.out.println(i);
10      }
11  }
```

Variable Scope

```
BUILD FAILURE
------------------------------------------------------------------------
Total time:  3.481 s
Finished at: 2022-11-11T08:42:45+08:00
------------------------------------------------------------------------
Failed to execute goal org.codehaus.mojo:exec-maven-plugin:3.0.0:exec (de
fault-cli) on project Demonstration: Command execution failed.: Process e
xited with an error: 1 (Exit value: 1) -> [Help 1]

To see the full stack trace of the errors, re-run Maven with the -e switc
h.
Re-run Maven using the -X switch to enable full debug logging.

For more information about the errors and possible solutions, please read
 the following articles:
[Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoExecutionEx
ception
```

5.1 Variable Scope

```
def sample(x):
    x[0] = 10
    x = [20, 30, 40]
    x[1] = 15
    print(x)


y = [5, 6, 7]
sample(y)
print(y)
```

Variable Scope

```
[20, 15, 40]
[10, 6, 7]
```

5.1 Variable Scope

```
y = [5, 6, 7]
sample(y)
print(y)


def sample(x):
    x[0] = 10
    x = [20, 30, 40]
    x[1] = 15
    print(x)
```

Variable Scope

```
Traceback (most recent call last):
  File "C:\Users\Dell\AppData\Local\Programs\
Python\Python310\demo.py", line 2, in <module
>
    sample(y)
NameError: name 'sample' is not defined
```

Variable Scope

```
Traceback (most recent call last):
  File "C:\Users\Dell\AppData\Local\Programs\
Python\Python310\demo.py", line 2, in <module
>
    sample(y)
NameError: name 'sample' is not defined
```

Variable Scope

```
1   package com.mycompany.demonstration;
2
3   public class Demonstration {
4
5       public static void main(String args[]){
6           hello();
7       }
8
9       static void hello(){
10          System.out.println("Hello Wrold");
11      }
12  }
```

Variable Scope

```
Building Demonstration 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ Demonstration ---
Hello Wrold
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.493 s
Finished at: 2022-11-11T08:53:46+08:00
------------------------------------------------------------------------
```

Compare and Contrast

- Always remember sequence structures and best practices

Compare and Contrast

```
print('Hello World')
```

```
Hello World
```

Compare and Contrast

```
1  class Demo{
2      public static void main(String args[]){
3          System.out.println("Hello World");
4      }
5  }
```

```
Hello World
```

Functions that Reuse Code

```
def hello():
    print("Hello World")

hello()
```

Functions that Reuse Code

```
1   package com.mycompany.demonstration;
2
3   public class Demonstration {
4       static void hello(){
5           System.out.println("Hello World");
6       }
7
8       public static void main(String args[]){
9           hello();
10      }
11  }
```

6.1 Compare and Contrast

Functions that Return Value

```
def hello():
    return "Hello World"


print(hello())
```

Functions that Return Value

```
package com.mycompany.demonstration;

public class Demonstration {
    static String hello(){
        return "Hello World";
    }

    public static void main(String args[]){
        System.out.println(hello());
    }
}
```

Functions that Return Value

```
def hello():
    print("Hello World")
    return "Hello World"


print(hello())
```

Functions that Return Value

```
Hello World
Hello World
```

Functions that Return Value

```
1   package com.mycompany.demonstration;
2
3   public class Demonstration {
4       static String hello(){
5           System.out.println("Hello World");
6           return "Hello World";
7       }
8
9       public static void main(String args[]){
10          System.out.println(hello());
11      }
12  }
```

Functions that Return Value