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**

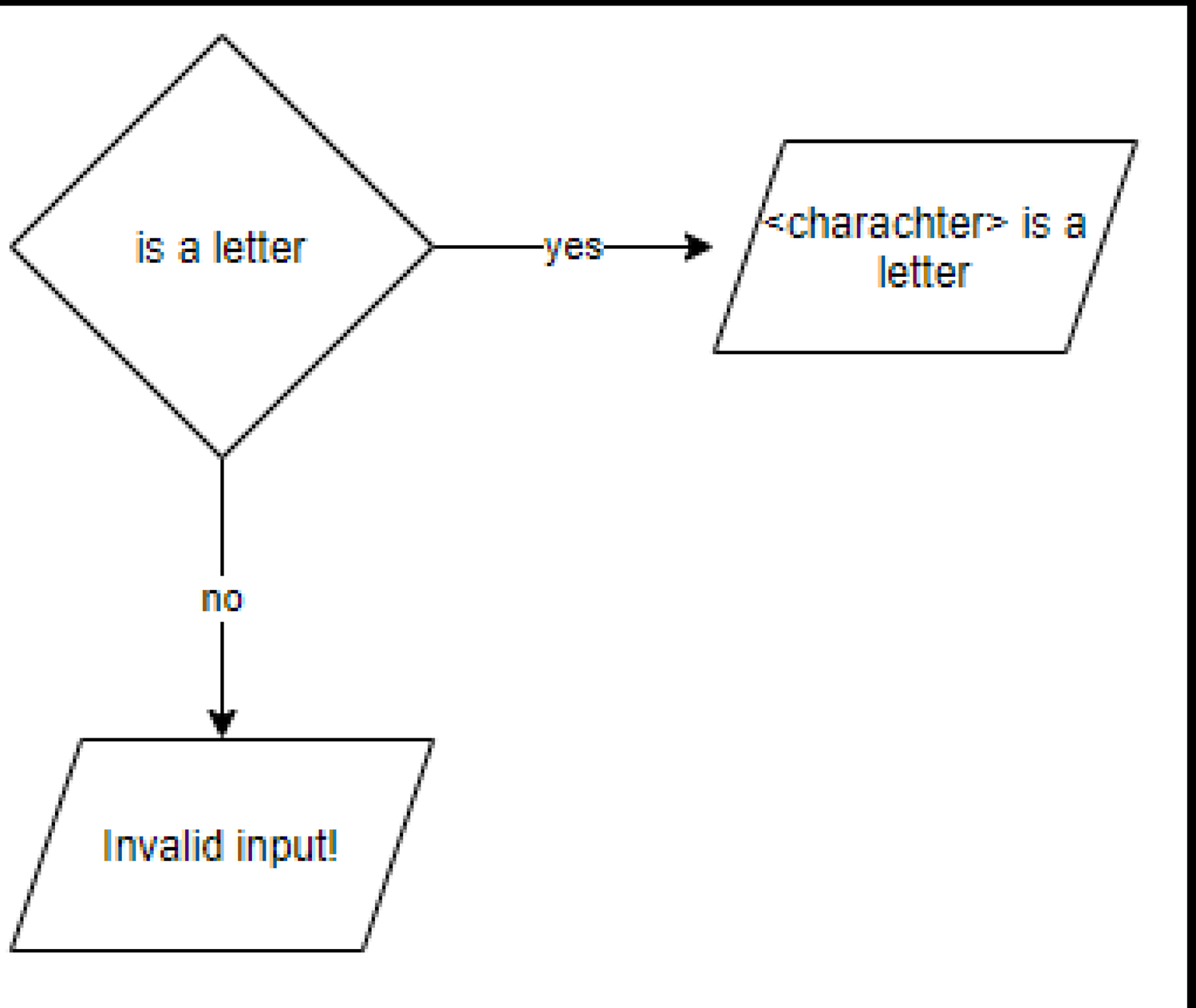
Definition

- 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.

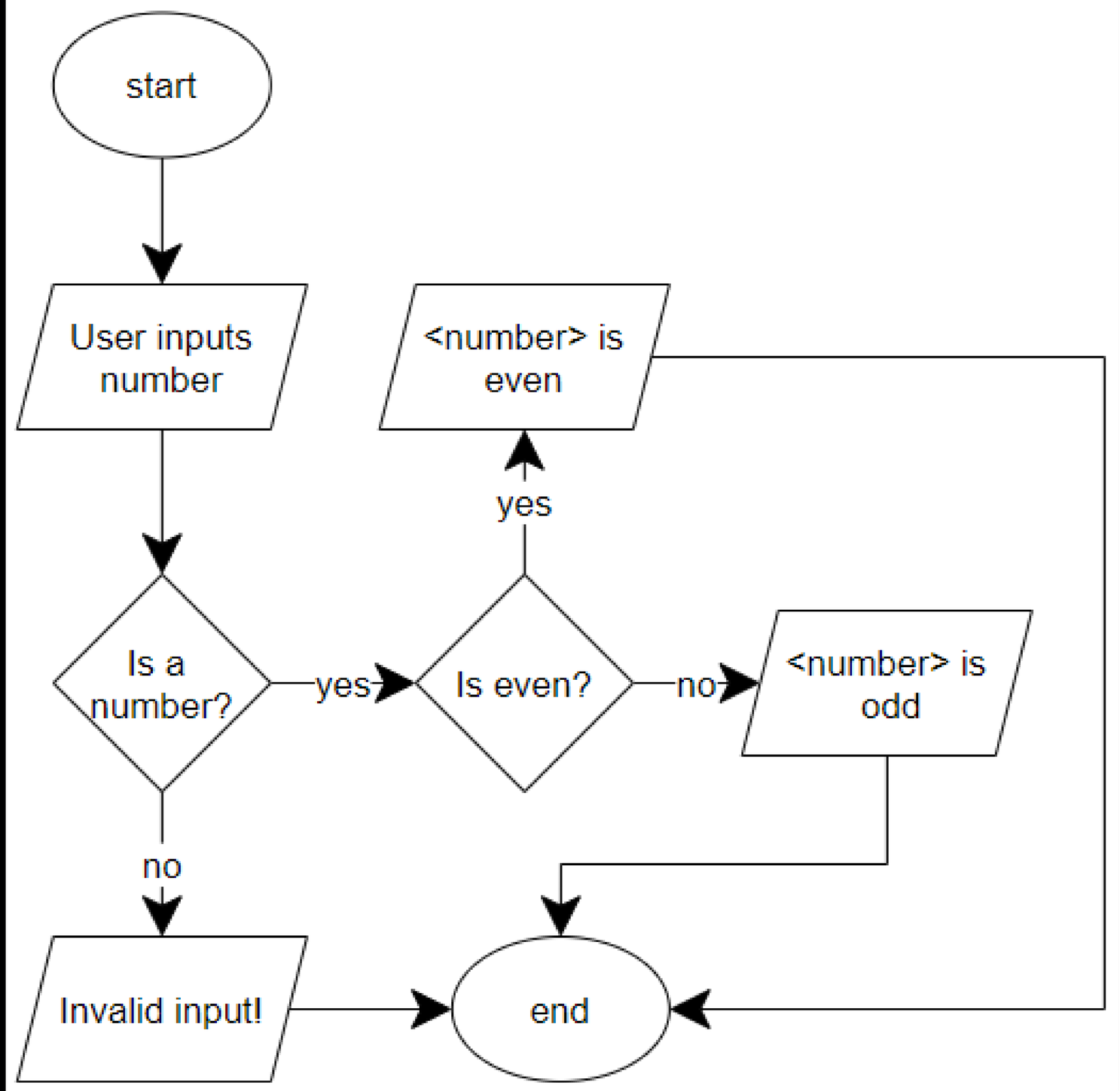
Definition

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

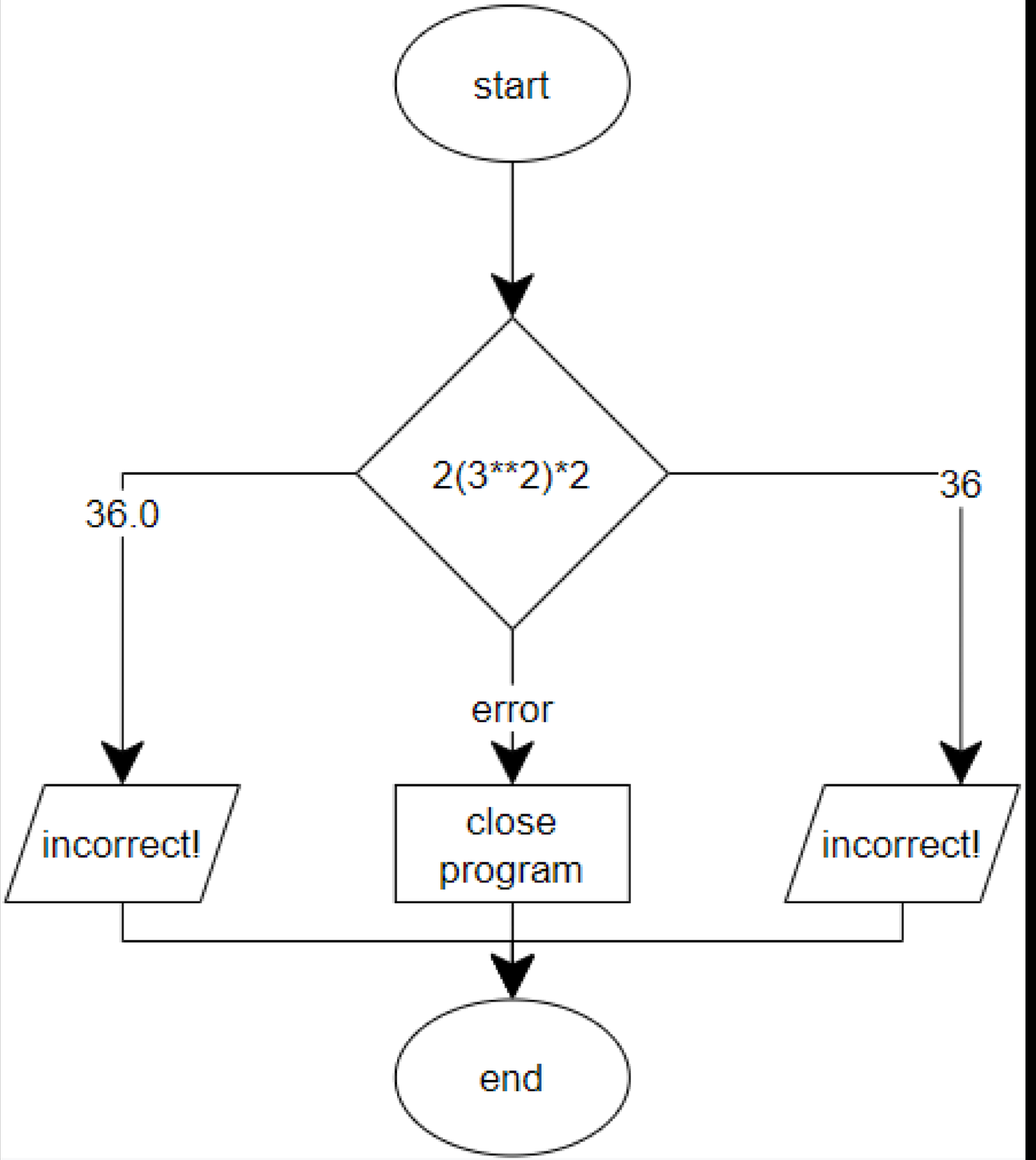
1.1 Conditions



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

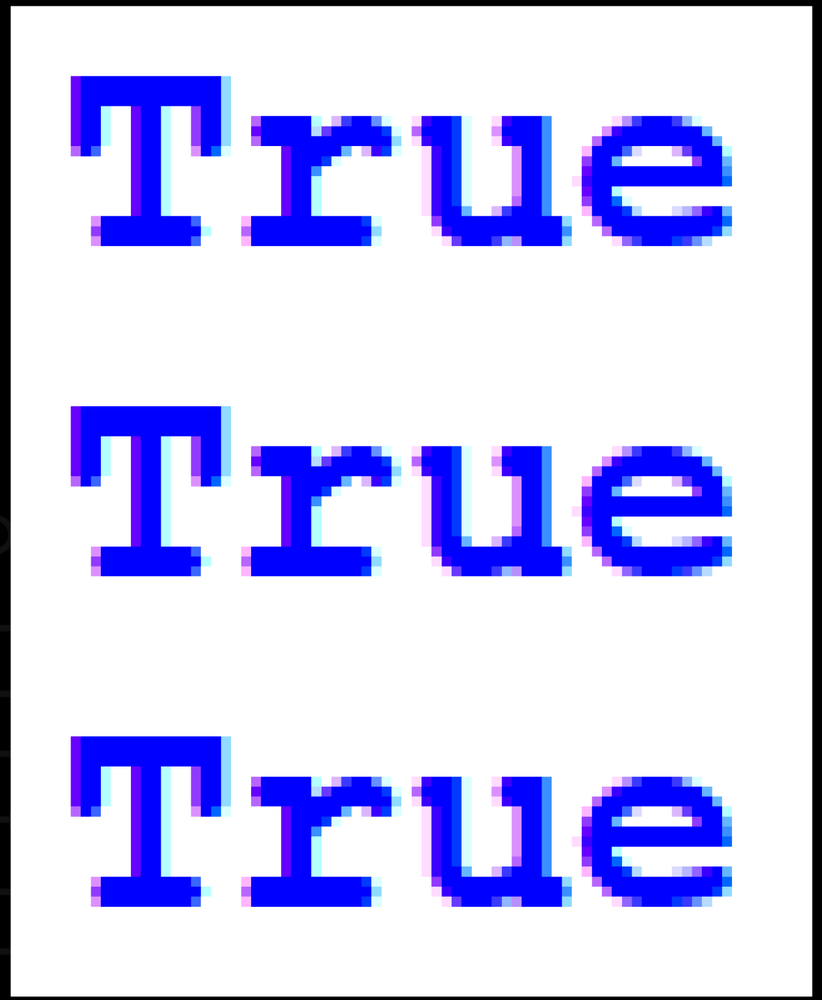
Boolean Algebra

- 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

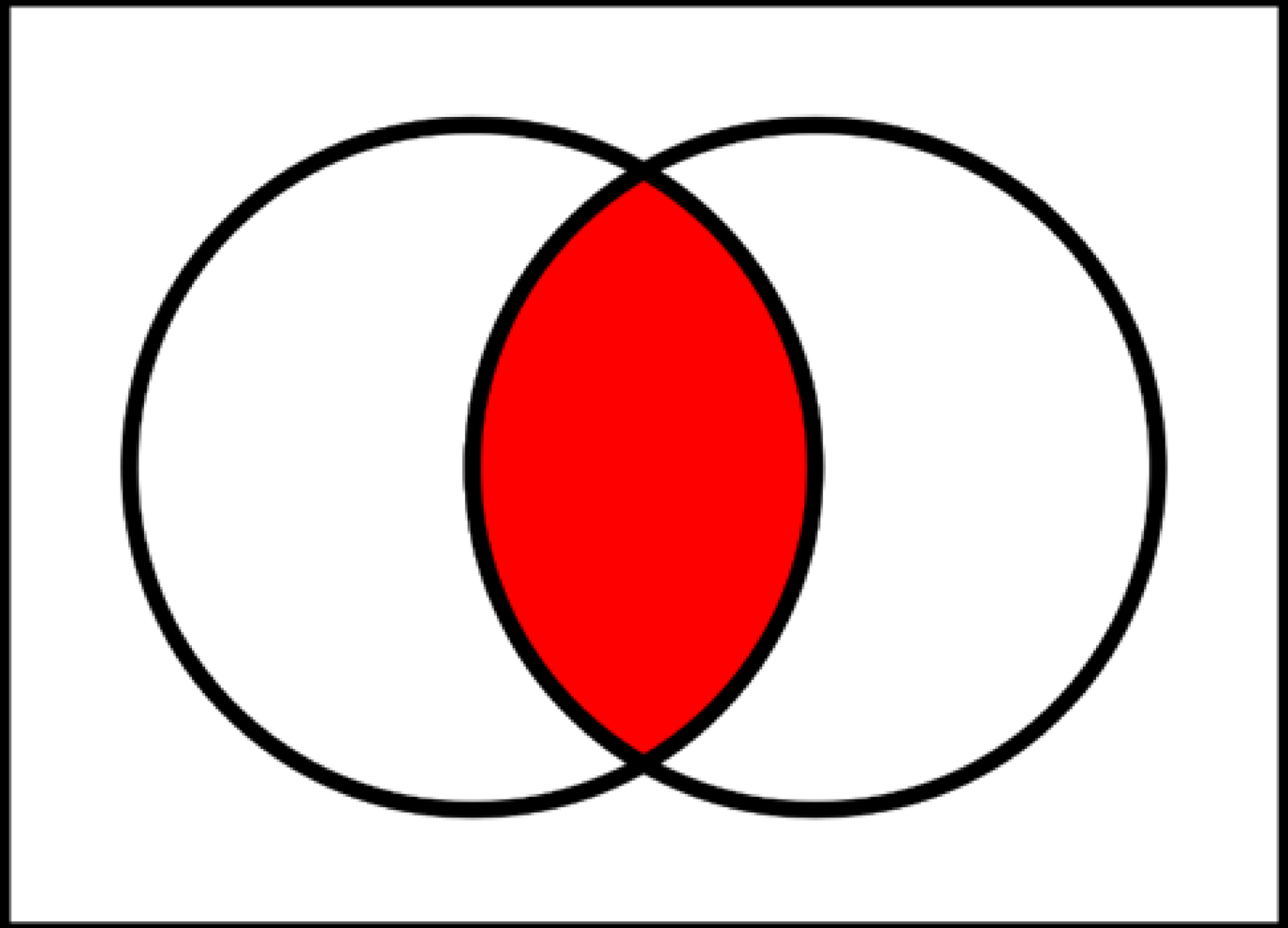
Conjunction

- 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

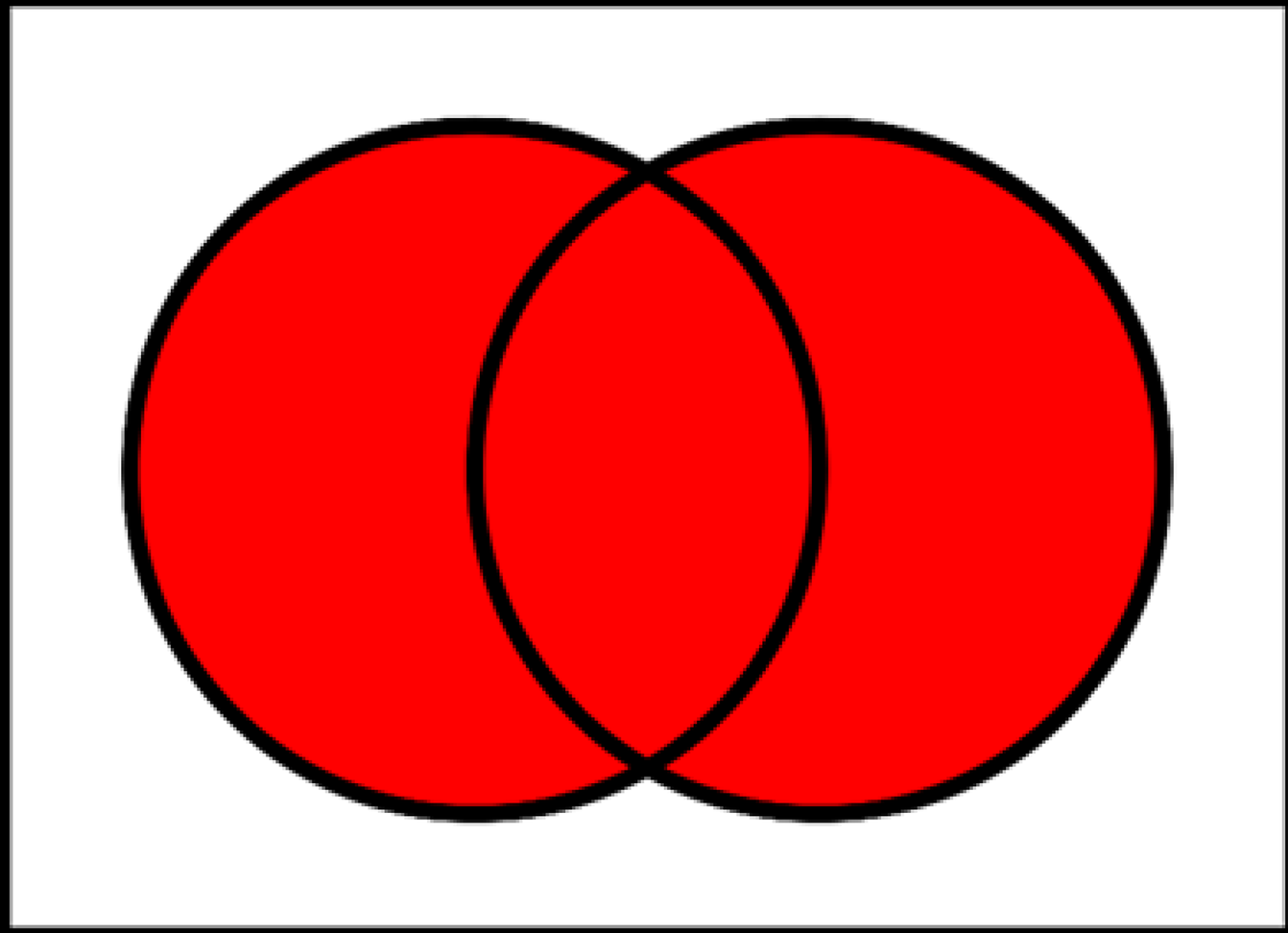
Inclusive

- 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

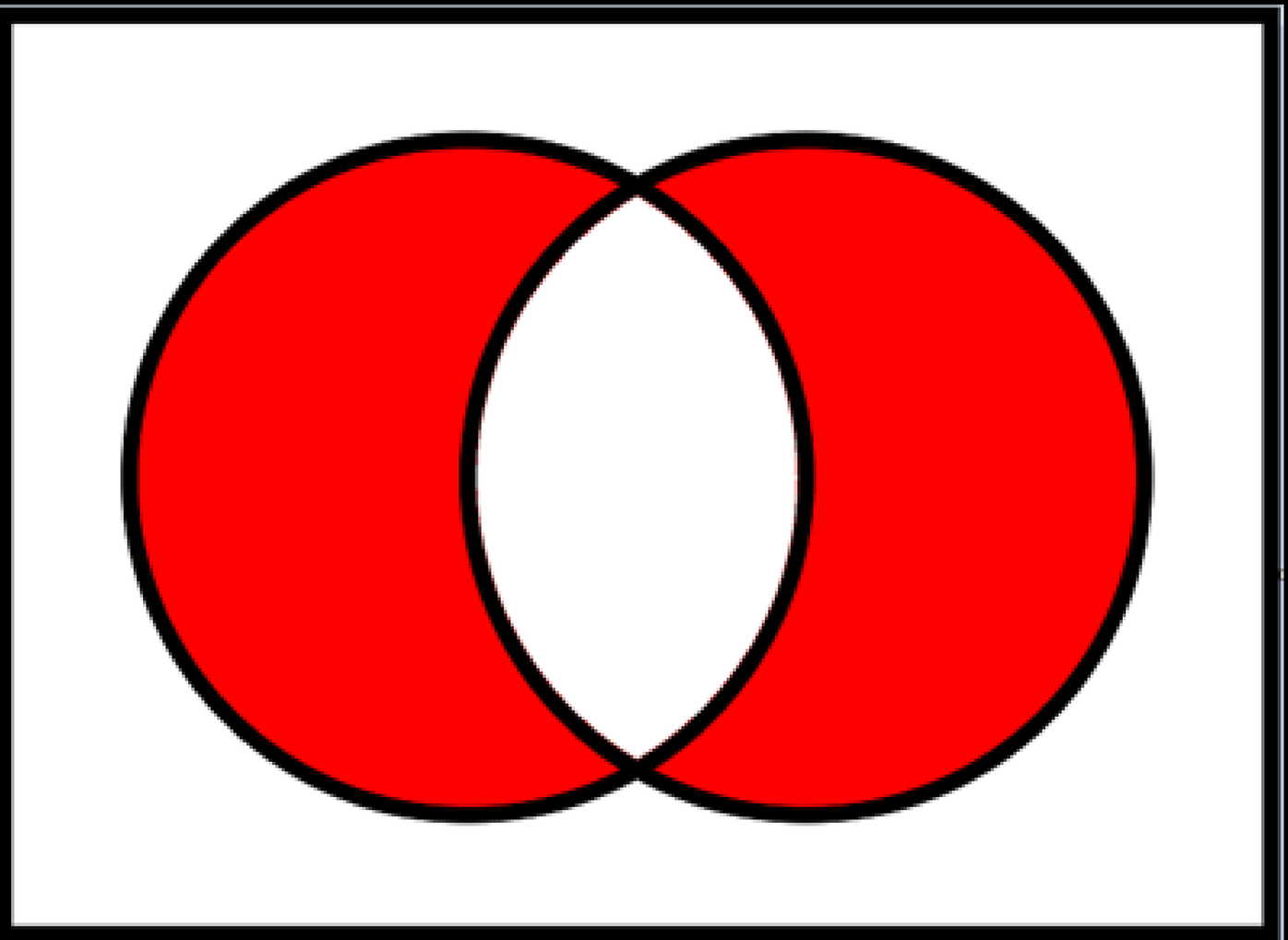
Exclusive

- 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

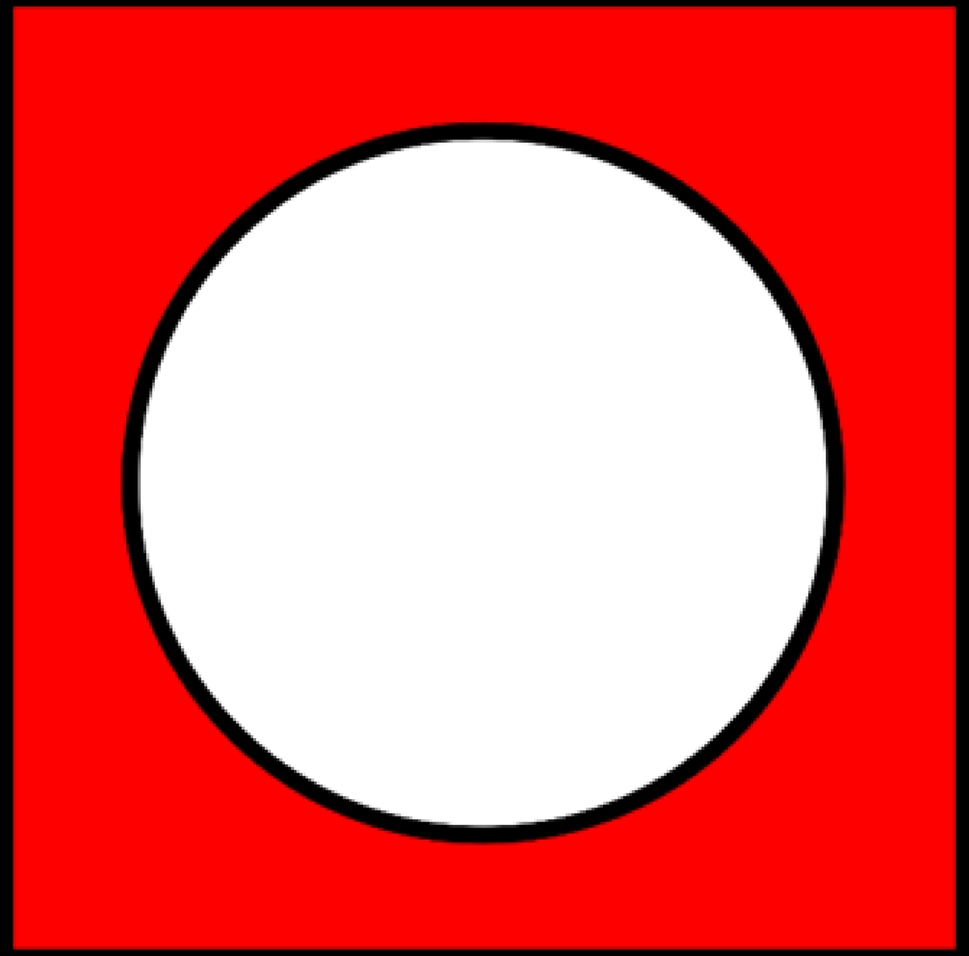
Negation

- 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

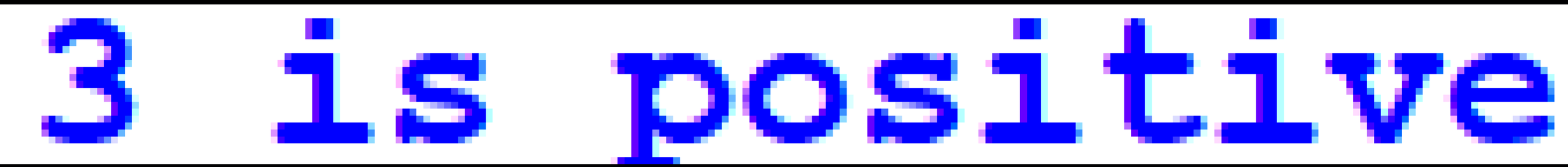
- can be consecutive
- syntax:

```
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')
```

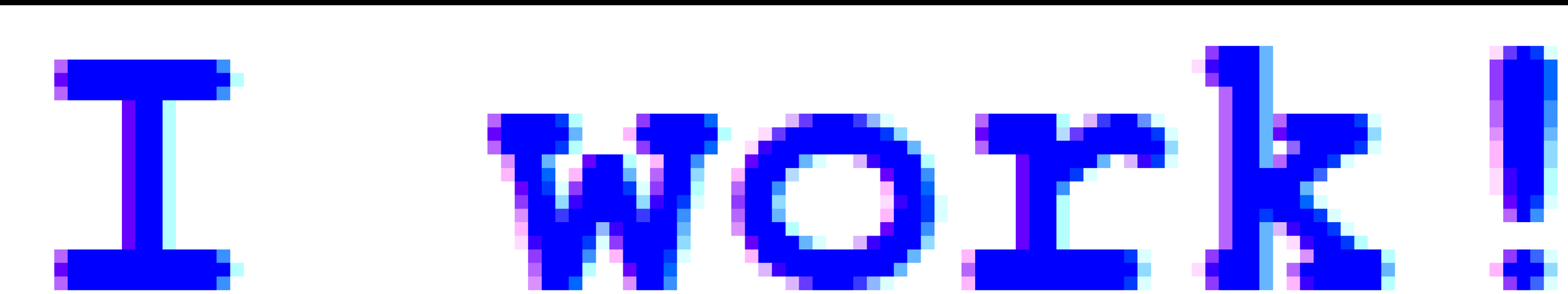
‘if’ Statement

```
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:

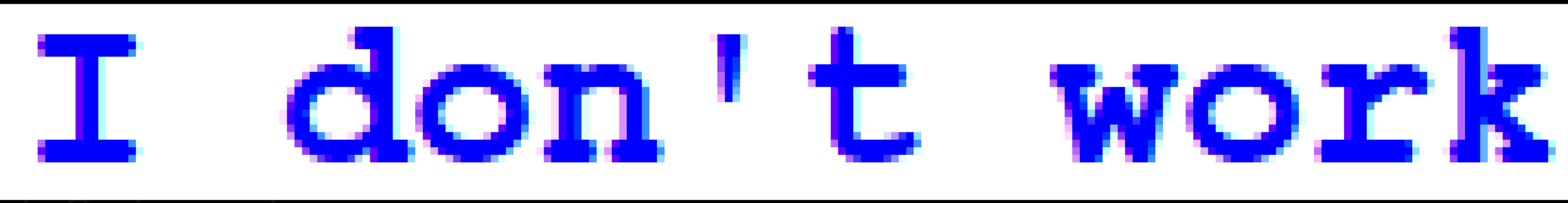
```
if(False):
        <content that will not run>
else:
        <content that will run>
```

‘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

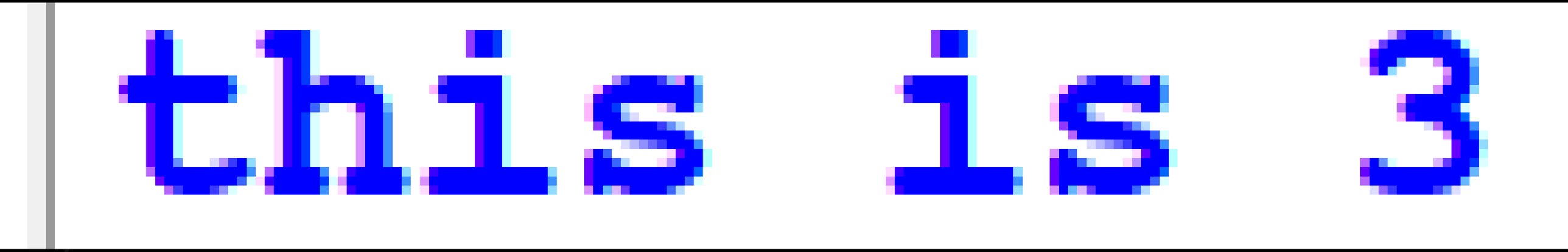
- syntax:

```
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')
```

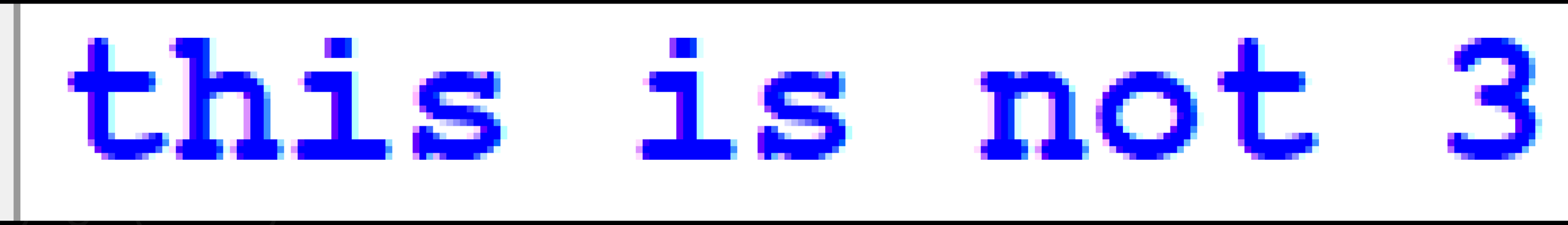
‘Match-Case’ Statement



‘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

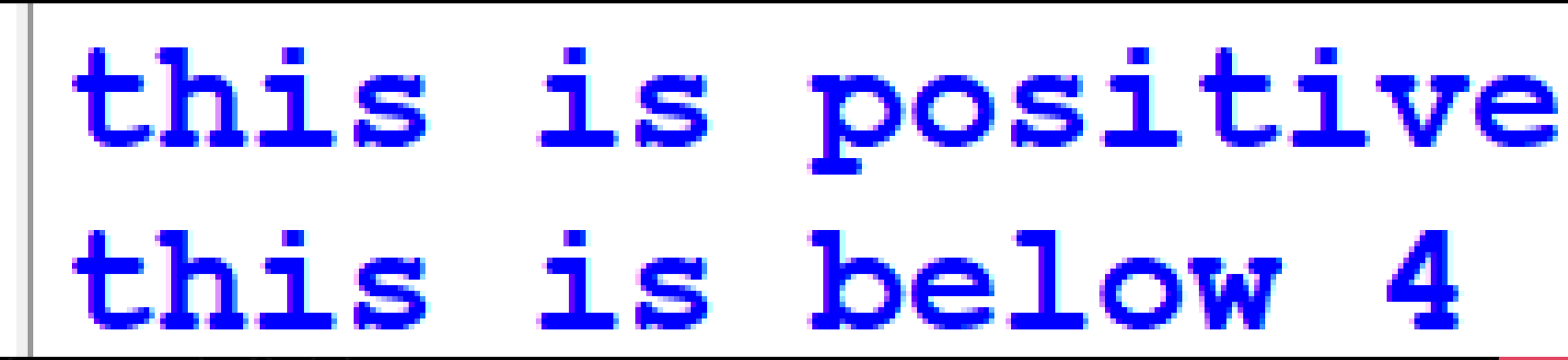
- if that is the case, can we put ‘if’ statements inside another ‘if’ statement? A ‘match-case’ inside an ‘if’ statement? An ‘if’ statement inside a ‘match-case’?

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