Introduction to

PYTHON PROGRAMING

CONDITIONAL LOGIC AND CONTROL FLOW

CONDITIONAL LOGIC IS BASED ON PERFORMING DIFFERENT ACTIONS DEPENDING ON WHETHER OR NOT SOME EXPRESSION, CALLED A CONDITIONAL, IS TRUE OR FALSE.

A STANDARD SET OF SYMBOLS CALLED BOOLEAN COMPARATORS ARE USED TO MAKE COMPARISONS

Boolean Comparator	Example	Meaning
>	a > b	a greater than b
<	a < b	a less than b
>=	a >= b	a greater than or equal to b
<=	a <= b	a less than or equal to b
!=	a != b	a not equal to b
==	a == b	a equal to b

THE TERM BOOLEAN IS DERIVED FROM THE LAST NAME OF THE ENGLISH MATHEMATICIAN GEORGE BOOLE, WHOSE WORKS HELPED LAY THE FOUNDATIONS OF MODERN COMPUTING.

CONDITIONAL LOGIC IS SOMETIMES CALLED **BOOLEAN LOGIC**, AND CONDITIONALS ARE SOMETIMES CALLED **BOOLEAN EXPRESSIONS**.

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

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

Note that `True` and `False` both start with capital letters.

The result of evaluating a conditional is always a boolean value:

```
>>> 1 == 1
True

>>> 3 > 5
False
```

Important

A common mistake when writing conditionals is to use the assignment operator `=`, instead of `==`, to test whether or not two values are equal.

Fortunately, Python will raise a `SyntaxError` if this mistake is encountered, so you'll know about it before you run your program.

REVIEW EXERCISES

1. For each of the following conditional expressions, guess whether they evaluate to `True` or `False`. Then type them into the interactive window to check your answers:

```
1 <= 1
1 != 1
1 != 2
"good" != "bad"
"good" != "Good"
123 == "123"
```

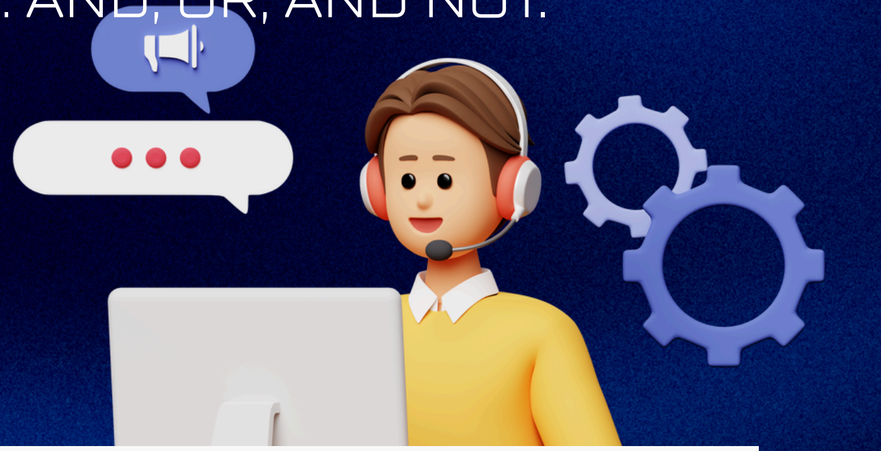
2. For each of the following expressions, fill in the blank (indicated by __) with an appropriate boolean comparator so that the expression evaluates to `True`:

```
3 __ 4
10 __ 5
"jack" __ "jill"
42 __ "42"
```

IN ADDITION TO BOOLEAN COMPARATORS, PYTHON HAS SPECIAL KEYWORDS CALLED LOGICAL OPERATORS THAT CAN BE USED TO COMBINE BOOLEAN EXPRESSIONS. THERE ARE THREE LOGICAL OPERATORS: AND, OR, AND NOT.

- **THE AND KEYWORD**

PYTHON'S AND OPERATOR WORKS EXACTLY THE SAME WAY.
HERE ARE FOUR EXAMPLE
OF COMPOUND STATEMENTS WITH AND:



```
>>> 1 < 2 and 3 < 4  # Both are True
True
```

Both statements are `True`, so the combination is also `True`.

```
>>> 2 < 1 and 4 < 3  # Both are False
False
```

Both statements are `False`, so their combination is also `False`.

```
>>> 1 < 2 and 4 < 3  # Second statement is False
False
```

```
>>> 2 < 1 and 3 < 4  # First statement is False
False
```

`2 < 1` is `False`, and `3 < 4` is `True`, so their combination is `False`.

- **THE OR KEYWORD**

WHEN WE USE THE WORD “OR” IN EVERYDAY CONVERSATION, SOMETIMES WE MEAN AN **EXCLUSIVE OR**. THAT IS, ONLY THE FIRST OPTION OR THE SECOND OPTION CAN BE TRUE.

```
>>> 1 < 2 or 3 < 4  # Both are True
True

>>> 2 < 1 or 4 < 3  # Both are False
False

>>> 1 < 2 or 4 < 3  # Second statement is False
True

>>> 2 < 1 or 3 < 4  # First statement is False
True
```

- **THE NOT KEYWORD**

THE NOT KEYWORD REVERSES THE TRUTH VALUE OF A SINGLE EXPRESSION:

```
>>> not True
False

>>> not False
True
```

ONE THING TO KEEP IN MIND WITH NOT, THOUGH, IS THAT IT DOESN'T ALWAYS BEHAVE THE WAY YOU MIGHT EXPECT WHEN COMBINED WITH COMPARATORS LIKE ==. FOR EXAMPLE, NOT TRUE == FALSE RETURNS TRUE, BUT FALSE == NOT TRUE WILL RAISE AN ERROR:

```
>>> not True == False
True

>>> False == not True
  File "<stdin>", line 1
    False == not True
             ^
SyntaxError: invalid syntax
```

BUILDING COMPLEX EXPRESSIONS

YOU CAN COMBINE THE AND, OR AND NOT KEYWORDS WITH TRUE AND FALSE TO CREATE MORE COMPLEX EXPRESSIONS

WHEN WORKING THROUGH COMPLICATED EXPRESSIONS, THE BEST STRATEGY IS TO START WITH THE MOST COMPLICATED PART OF THE EXPRESSION AND BUILD OUTWARD FROM THERE.

REVIEW EXERCISES

1. Figure out what the result will be (`True` or `False`) when evaluating the following expressions, then type them into the interactive window to check your answers:

```
(1 <= 1) and (1 != 1)
not (1 != 2)
("good" != "bad") or False
("good" != "Good") and not (1 == 1)
```

2. Add parentheses where necessary so that each of the following expressions evaluates to `True`:

```
False == not True
True and False == True and False
not True and "A" == "B"
```

CONTROL THE FLOW OF YOUR PROGRAM

- ## THE IF STATEMENT

AN IF STATEMENT TELLS PYTHON TO ONLY EXECUTE A PORTION OF CODE IF A CONDITION IS MET.

```
if 2 + 2 == 4:
    print("2 and 2 is 4")
```

JUST LIKE WHILE LOOPS, AN IF STATEMENT HAS THREE PARTS:

1. THE IF KEYWORD
2. A TEST CONDITION, FOLLOWED BY A COLON
3. AN INDENTED BLOCK OF CODE THAT IS EXECUTED IF THE TEST CONDITION IS TRUE

Note

Leaving off the colon (`:`) after the test condition in an `if` statement raises a `SyntaxError`:

```
>>> if 2 + 2 == 4
SyntaxError: invalid syntax
```

THE ELSE KEYWORD

THE ELSE KEYWORD IS USED AFTER AN IF STATEMENT IN ORDER TO EXECUTE SOME CODE ONLY IF THE IF STATEMENT'S TEST CONDITION IS FALSE.

```
grade = 40

if grade >= 70:
    print("You passed the class!")
else:
    print("You did not pass the class :(")

print("Thank you for attending.")
```

Important

Leaving off the colon (:) from the `else` keyword will raise a `SyntaxError`:

```
>>> if 2 + 2 == 5:
...     print("Who broke my math?")
... else
SyntaxError: invalid syntax
```

THE ELIF KEYWORD

THE ELIFKEYWORDISSHORTFOR“ELSEIF”ANDCANBEUSEDTOADDADDITIONAL CONDITIONS AFTER AN IF STATEMENT

JUST LIKE IF STATEMENTS, ELIF STATEMENTS HAVE THREE PARTS:

1. THE ELIF KEYWORD
2. ATEST CONDITION, FOLLOWED BY A COLON
3. ANINDENTED CODE BLOCK THAT IS EXECUTED IF THE TEST CONDITION EVAL UATES TO TRUE

Important

Leaving off the colon (:) at the end of an `elif` statement raises a `SyntaxError`:

```
>>> if 2 + 2 == 5:
...     print("Who broke my math?")
... elif 2 + 2 == 4
SyntaxError: invalid syntax
```

NESTED IF STATEMENTS

JUSTLIKEFORANDWHILELOOPSCANBENESTEDWITHINONEANOTHER,YOUNEST ANIFSTATEMENTINSIDE ANOTHERTOCREATECOMPLICATED DECISION MAKING STRUCTURES.

Note

The complexity that results from using deeply nested if statements may make it difficult to predict how your program will behave under given conditions.

For this reason, nested `if` statements are generally discouraged.

REVIEW EXERCISES

1. WRITE A SCRIPT THAT PROMPTS THE USER TO ENTER A WORD USING THE INPUT() FUNCTION, STORES THAT INPUT IN A VARIABLE, AND THEN DISPLAYS WHETHER THE LENGTH OF THAT STRING IS LESS THAN 5 CHARACTERS, GREATER THAN 5CHARACTERS, OR EQUAL TO 5 CHARACTERS BY USING A SET OF IF, ELIF AND ELSE STATEMENTS.

IF STATEMENTS AND FOR LOOPS

THE BLOCK OF CODE IN A FOR LOOP IS JUST LIKE ANY OTHER BLOCK OF CODE. THAT MEANS YOU CAN NEST AN IF STATEMENT IN A FOR LOOP JUST LIKE YOU CAN ANYWHERE ELSE IN YOUR CODE

```
sum_of_evens = 0


for n in range(1, 100):
    if n % 2 == 0:
        sum_of_evens = sum_of_evens + n


print(sum_of_evens)
```

BREAK

THE BREAK KEYWORD TELLS PYTHON TO LITERALLY BREAK OUT OF A LOOP. THAT IS, THE LOOP STOPS COMPLETELY AND ANY CODE AFTER THE LOOP IS EXECUTED.

```
for n in range(0, 4):
    if n == 2:
        break
    print(n)

print(f"Finished with n = {n}")
```

Only the first two numbers are printed in the output:

```
0
1
Finished with n = 2
```

CONTINUE

THE CONTINUE KEYWORD IS USED TO SKIP ANY REMAINING CODE IN THE LOOP BODY AND CONTINUE ON TO THE NEXT ITERATION

```
for i in range(0, 4):
    if i == 2:
        continue
    print(i)

print(f"Finished with i = {i}")
```

All the numbers except for 2 are printed in the output:

```
0
1
3
Finished with i = 3
```

Note

It's always a good idea to give short but descriptive names to your variables that make it easy to tell what they are supposed to represent.

The letters `i`, `j` and `k` are exceptions because they are so common in programming.

These letters are almost always used when we need a "throw-away" number solely for the purpose of keeping count while working through a loop.

REVIEW EXCERCISES:

1.USING BREAK, WRITE A PROGRAM THAT REPEATEDLY ASKS THE USER FOR SOME INPUT AND ONLY QUITS IF THE USER ENTERS "Q" OR "Q".

2. USING CONTINUE, WRITE A PROGRAM THAT LOOPS OVER THE NUMBERS 1 TO 50 AND PRINTS ALL NUMBERS THAT ARE NOT MULTIPLES OF 3.

RECOVER FROM ERRORS

- *VALUE ERROR* OCCURS WHEN AN OPERATION ENCOUNTERS AN INVALID VALUE.

```
>>> int("not a number")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int("not a number")
ValueError: invalid literal for int() with base 10: 'not a number'
```

- A *TYPE ERROR* OCCURS WHEN AN OPERATION IS PERFORMED ON A VALUE OF THE WRONG TYPE.

```
>>> "1" + 2
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    "1" + 2
TypeError: can only concatenate str (not "int") to str
```

- A *NAME ERROR* OCCURS WHEN YOU TRY TO USE A VARIABLE NAME THAT HASN'T BEEN DEFINED YET:

```
>>> print(does_not_exist)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(does_not_exist)
NameError: name 'does_not_exist' is not defined
```

- A *ZERO DIVISION ERROR* OCCURS WHEN THE DIVISOR IN A DIVISION OPERATION IS 0:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    1 / 0
ZeroDivisionError: division by zero
```

- AN *OVERFLOW ERROR* OCCURS WHEN THE RESULT OF AN ARITHMETIC OPERATION IS TOO LARGE.

```
>>> pow(2.0, 1_000_000)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    pow(2.0, 1_000_000)
OverflowError: (34, 'Result too large')
```

THE *TRY* AND *EXCEPT* KEYWORDS

THE *TRY* KEYWORD IS USED TO INDICATE A *TRY* BLOCK AND IS FOLLOWED BY A COLON. THE CODE INDENTED AFTER *TRY* IS EXECUTED.

```
try:
    number = int(input("Enter an integer: "))
except ValueError:
    print("That was not an integer")
```

THE“BARE” *EXCEPT* CLAUSE

```
try:
    # Do lots of hazardous things that might break
except:
    print("Something bad happened!")
```

IF ANY EXCEPTION IS RAISED WHILE EXECUTING THE CODE IN THE *TRY* BLOCK, THE EXCEPT BLOCK WILL RUN AND THE MESSAGE "SOMETHING BAD HAPPENED!" WILL BE DISPLAYED.

REVIEW EXCERCISE:

1. WRITE A SCRIPT THAT REPEATEDLY ASKS THE USER TO INPUT AN INTEGER, DISPLAYING A MESSAGE TO “TRY AGAIN" BY CATCHING THE VALUEERROR THAT IS RAISED IF THE USER DID NOT ENTER AN INTEGER. ONCE THE USER ENTERS AN INTEGER, THE PROGRAM SHOULD DISPLAY THE NUMBER BACK TO THE USER AND END WITHOUT CRASHING.

2. WRITE A PROGRAM THAT ASKS THE USER TO INPUT A STRING AND AN INTEGER N. THEN DISPLAY THE CHARACTER AT INDEX N IN THE STRING.

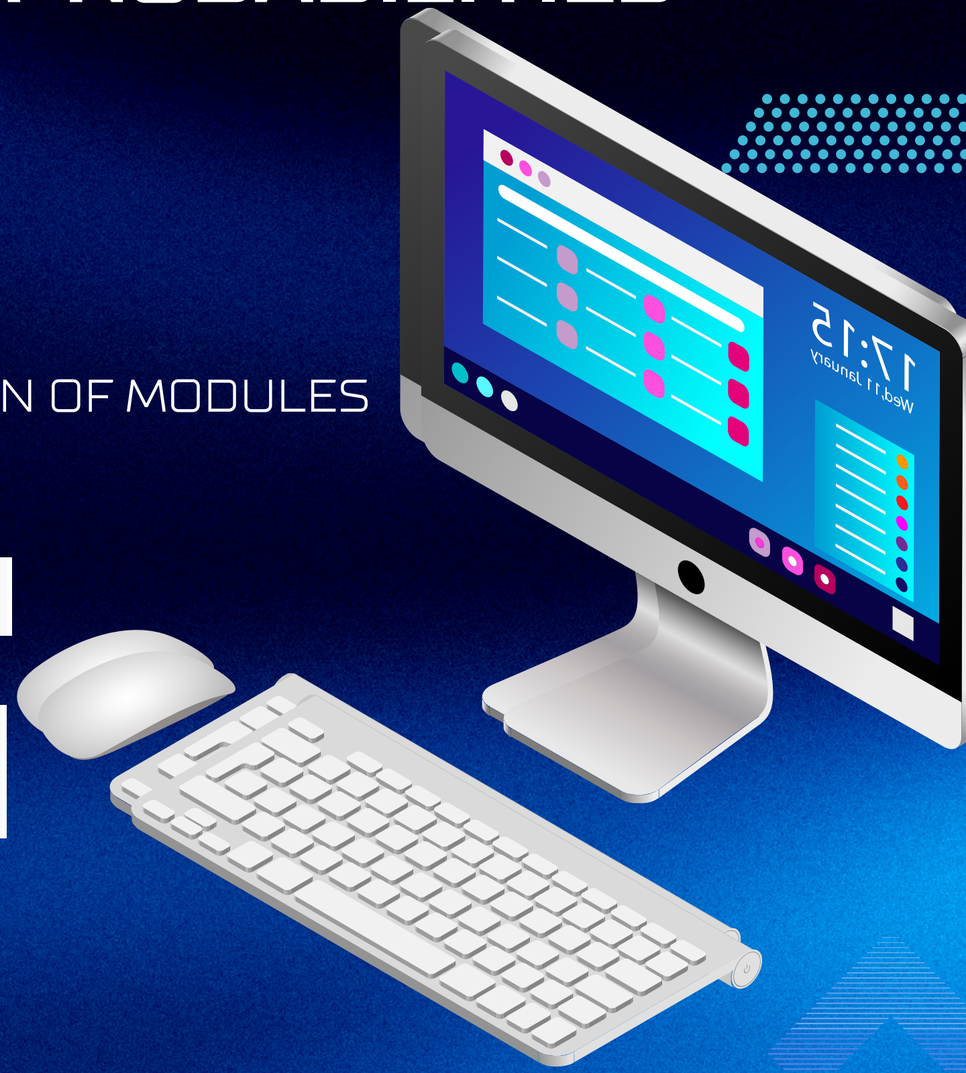
SIMULATE EVENTS AND CALCULATE PROBABILITIES

THE *RANDOM* MODULE

A **MODULE** IS A COLLECTION OF RELATED CODE.
PYTHON'S **STANDARD LIBRARY** IS AN ORGANIZED COLLECTION OF MODULES THAT YOU CAN **IMPORT**.

```
>>> import random
```

```
>>> random.randint(1, 10)
9
```



FLIPPING FAIR COINS

```
import random

def coin_flip():
    """Randomly return 'heads' or 'tails'."""
```

```
    if random.randint(0, 1) == 0:
        return "heads"
    else:
        return "tails"
```

```
# First initialize the tallies to 0
heads_tally = 0
tails_tally = 0

for trial in range(10_000):
    if coin_flip() == "heads":
        heads_tally = heads_tally + 1
    else:
        tails_tally = tails_tally + 1
```

TOSSING UNFAIR COINS

```python
import random

def unfair_coin_flip(probability_of_tails):
    if random.random() < probability_of_tails:
        return "tails"
    else:
        return "heads"
```

```python
heads_tally = 0
tails_tally= 0

for trial in range(10_000):
    if unfair_coin_flip(.7) == "heads":
        heads_tally = heads_tally + 1
    else:
        tails_tally = tails_tally + 1

ratio = heads_tally / tails_tally
print(f"The ratio of heads to tails is {ratio}")
```

REVIEW EXERCISES

1. WRITE A FUNCTION CALLED ROLL() THAT USES THE RANDINT() FUNCTION TO SIMULATE ROLLING A FAIR DIE BY RETURNING A RANDOM INTEGER BETWEEN 1 AND 6.

2. WRITE A SCRIPT THAT SIMULATES 10,000 ROLLS OF A FAIR DIE AND DISPLAYS THE AVERAGE NUMBER ROLLED.

CHALLENGE: SIMULATE AN ELECTION

SUPPOSE TWO CANDIDATES, CANDIDATE A AND CANDIDATE B, ARE RUNNING FOR MAYOR IN A CITY WITH THREE VOTING REGIONS. THE MOST RECENT POLLS SHOW THAT CANDIDATE A HAS THE FOLLOWING CHANCES FOR WINNING IN EACH REGION:

- REGION 1: 87%CHANCE OF WINNING
- REGION 2: 65%CHANCE OF WINNING
- REGION 3: 17% CHANCE OF WINNING

THANK YOU