MR2020: Coding for METOC

Module 6: Control Flows and List Comprehensions

Control flows are a vital part of programming in any language. They handle the logical progression of code. The main types of control flows we will use are

Conditional control flows:

if-elif-else	Common type of conditional block; For example, "if X, then do Y, but otherwise, do Z".
match-case	A more versatile version of if-elif-else; For example, if X looks like A then do B but if it looks like C then do D.

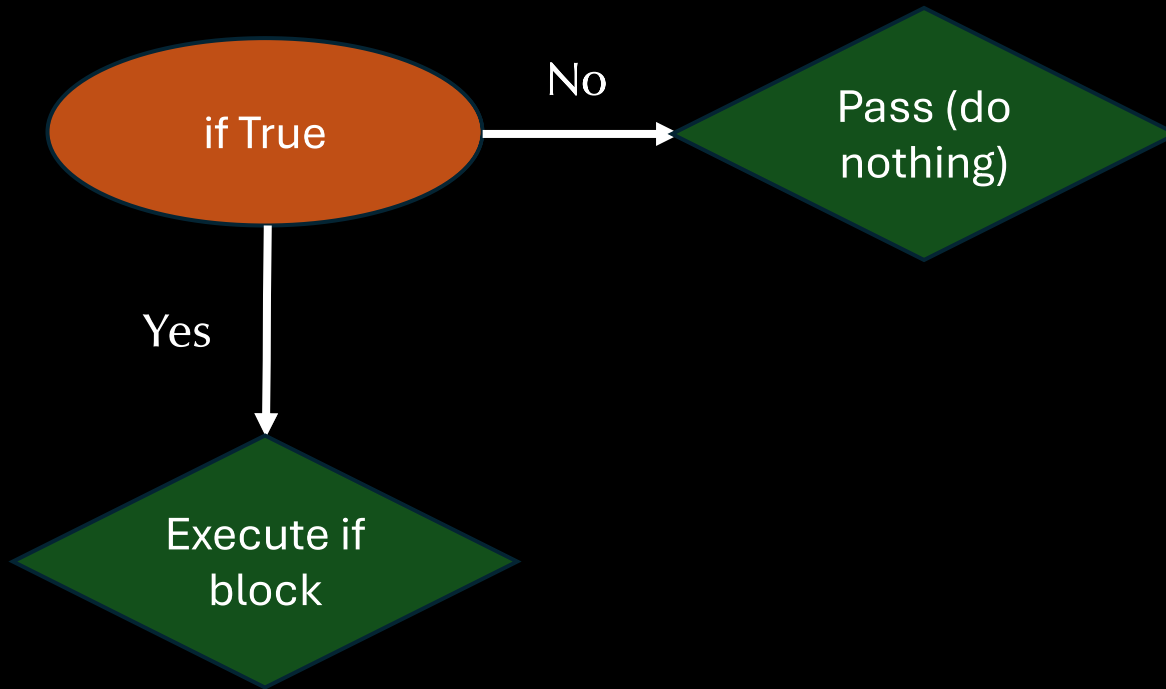
Exception handling:

try-except-else-finally	Useful when running code that may throw an exception. For example, try some code but if it fails execute a different code.

Looping control flows:

for loops	Execute code over all elements in an iterable object.
while loops	May execute indefinitely until a condition is not met.

if statement



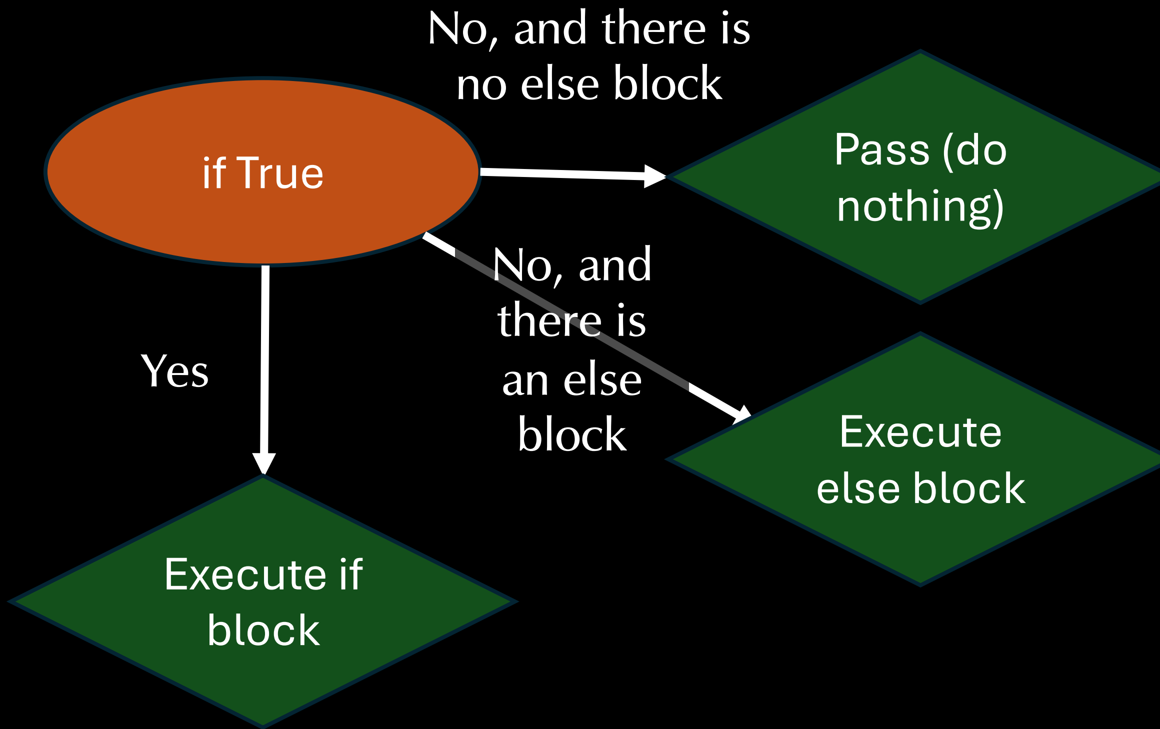
Example (nothing happens)

```
A = 3
if A == 4:
    print('A is 4')
```

Example (if statement is executed)

```
A = 3
if A < 4:
    print(‘A is less than 4.')
```

if-(else) block



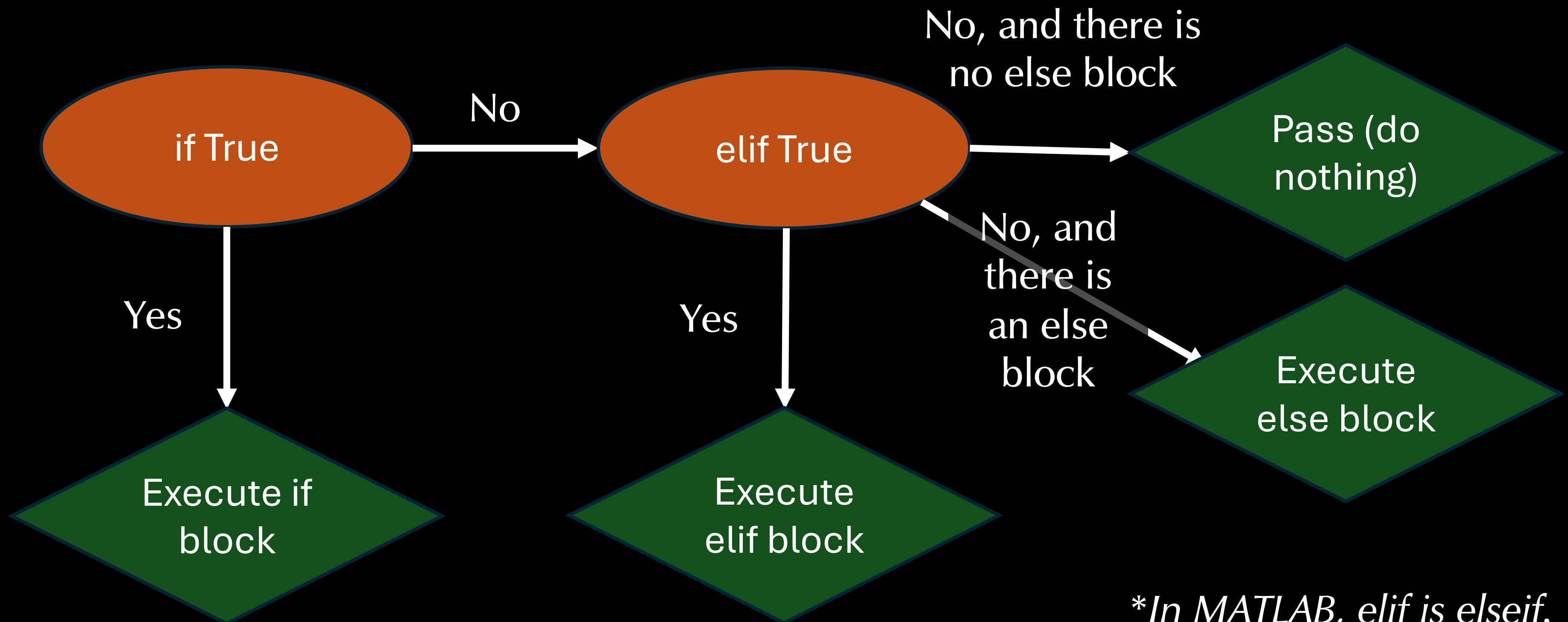
Example without else
(nothing happens)

```
A = 3
if A == 4:
    print('A is 4')
```

Example with else (last statement prints)

```
A = 3
if A == 4:
    print('A is 4')
else:
    print('A is not 4.')
```

if-elif-(else) block*



In MATLAB, elif is elseif.

Example without else (nothing happens)

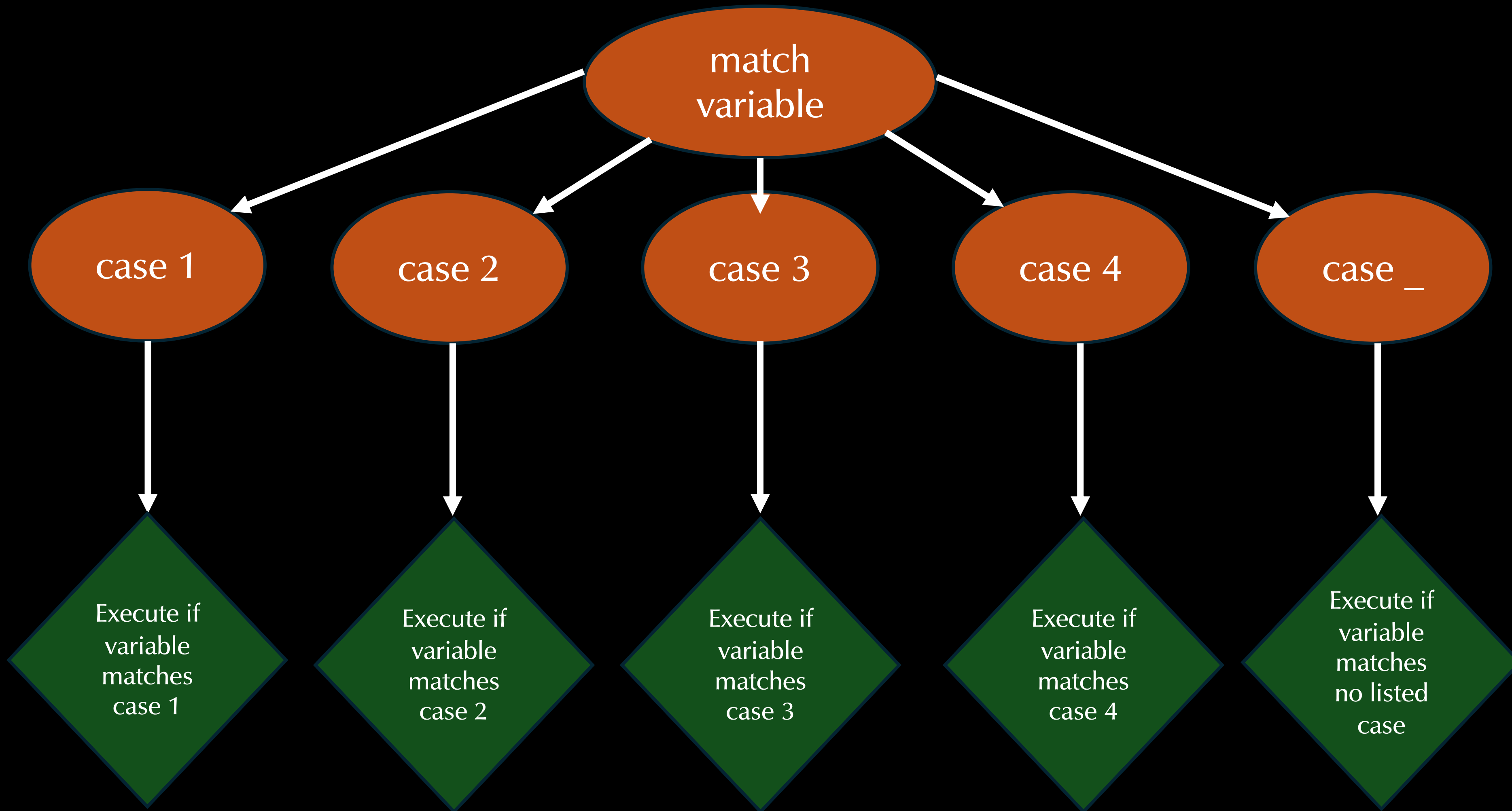
```
A = 3
if A == 4:
    print('A is 4')
elif A == 5:
    print('A is 5')
```

Example with else (last statement prints)

```
A = 3
if A == 4:
    print('A is 4')
elif A == 5:
    print('A is 5')
else:
    print('A is neither 4 nor 5.')
```

Match-case blocks

These function much like if-elif-else, but have more flexibility for handling data structures, better readability, and scalability. Useful for particularly long if-elif blocks if evaluating one variable. In MATLAB, this is known as switch-case.



A simple example of match-case and if-elif-else that do the same thing

```python
def describe_number_if(n):
    if n < 0:
        return "Negative number"
    elif n == 0:
        return "Zero"
    elif n > 0:
        return "Positive number"
```

```python
def describe_number_match(n):
    match n:
        case x if x < 0:
            return "Negative number”
        case 0:
            return "Zero"
        case x if x > 0:
            return "Positive number"
```

These two do exactly the same thing. In this case, there is little advantage to one over the other though.

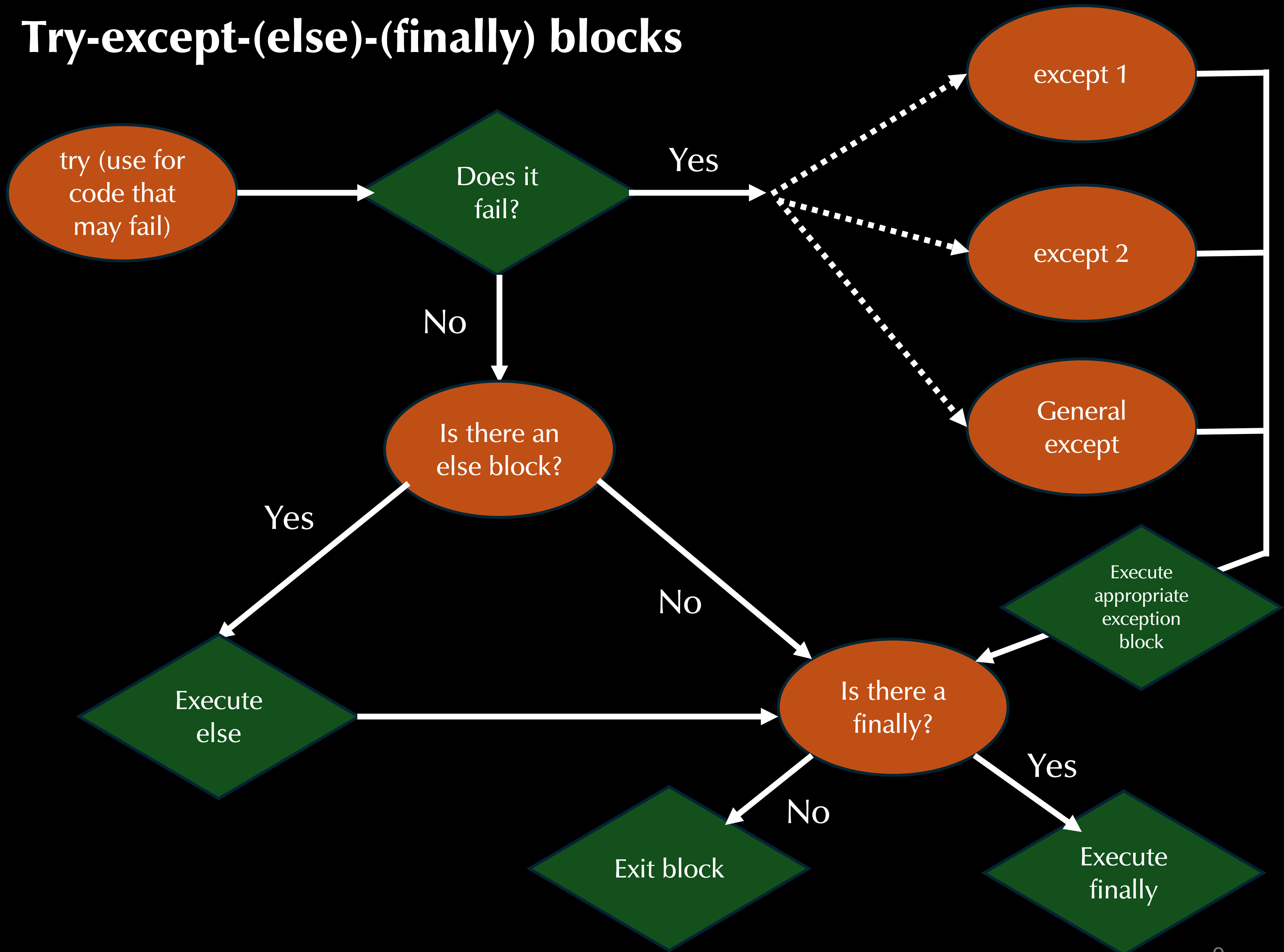
More complex example

```
data = [2,3]

# Match-case block
def processmatch(data):
    match data:
        case [x, y]: # Is data a 2-item list?
            return f"List with two elements: {x}, {y}"
        case (x, y, z): # Is data a 3-item tuple?
            return f"Tuple with three elements: {x}, {y}, {z}"
        case {'name': name, 'age': age}: # Is data a dictionary w/ name and age keys?
            return f"Dictionary with name and age: {name}, {age}"
        case _: # Is data none of the above?
            return "Unknown data structure"

# Equivalent if-elif-else block
def processif(data):
    # Is data a 2-item list?
    if isinstance(data, list) and len(data) == 2:
        x, y = data
        return f"List with two elements: {x}, {y}”
    # Is data a 3-item tuple?
    elif isinstance(data, tuple) and len(data) == 3:
        x, y, z = data
        return f"Tuple with three elements: {x}, {y}, {z}”
    # Is data a dictionary w/ name and age keys?
    elif isinstance(data, dict) and 'name' in data and 'age' in data:
        name = data['name’]
        age = data['age’]
        return f"Dictionary with name and age: {name}, {age}"
    else: # Is data none of the above?
        return "Unknown data structure"
```

Try-except-(else)-(finally) blocks



Try-except-(else)-(finally) example

```
A, B = 1, 0
try: # Divide A by B.
    C = A/B
except ZeroDivisionError: # If a divide by zero error occurs, do this.
    print('No dividing by zero!')
except TypeError: # If a TypeError occurs, do this.
    print('A type error has occurred.')
except: # If some other error occurs, do this instead.
    print('Something unknown went wrong!')
else: # Do this if the try code worked.
    print('You successfully divided!')
finally: # Do this regardless of what happened above.
    del A, B
```

Try running this code. See what happens if you change B to not be zero. What happens to A and B after the code is run?

Pass, continue, and break

`pass` is a null placeholder. It indicates that no action is to be taken. While technically not necessary, it can be used in a location (such as a function, class, or within any control flow) where future code is planned.

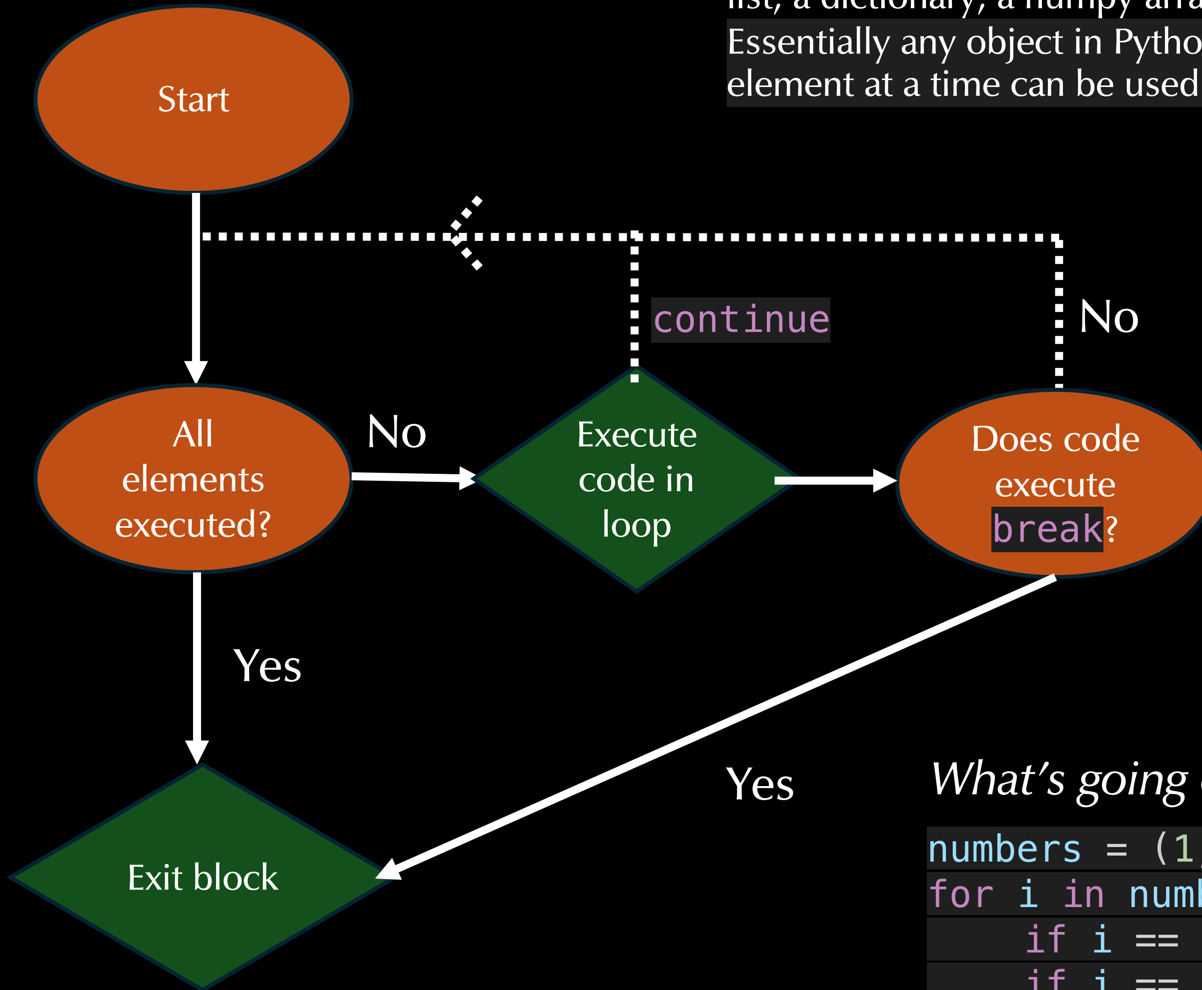
```
if A > 3:
    print('A is huge.’)
else: # Figure out what to do later.
    pass # This isn’t required. We could simply leave out the else block.
```

A couple of Python statements are particularly important in looping control flows.

`continue` statements will immediately stop the current iteration of a loop and return the loop back to the beginning. If in a for loop, it will iterate over the next element in the iterable.

`break` statements will cause the loop to immediately end, i.e., exit the loop and start executing whatever code (if any) follows the loop.

For loops

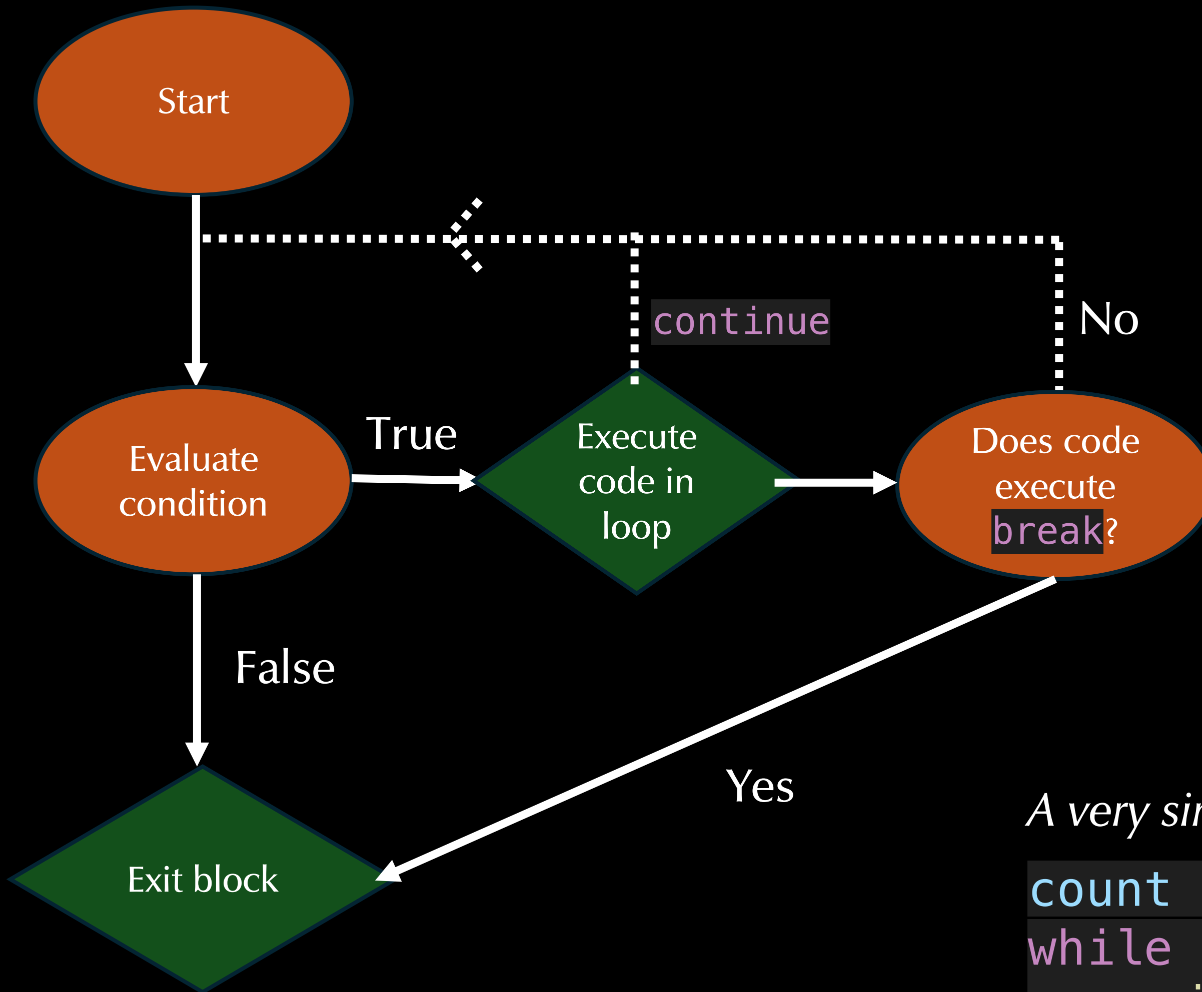


This (numbers) is just an example of an iterable. It could be a list, a dictionary, a numpy array, or a `range`, for example. Essentially any object in Python that can be accessed one element at a time can be used as an iterable.

What's going on below?

```
numbers = (1,2,3,4,5)
for i in numbers:
    if i == 4: continue
    if i == 5:
        print('Stopping loop...')
        break
    print(i)
```

While loops



A very simple while loop

```
count = 0
while count <= 4:
    print(count)
    count += 1
```

For loops are slow! Use them only as necessary!

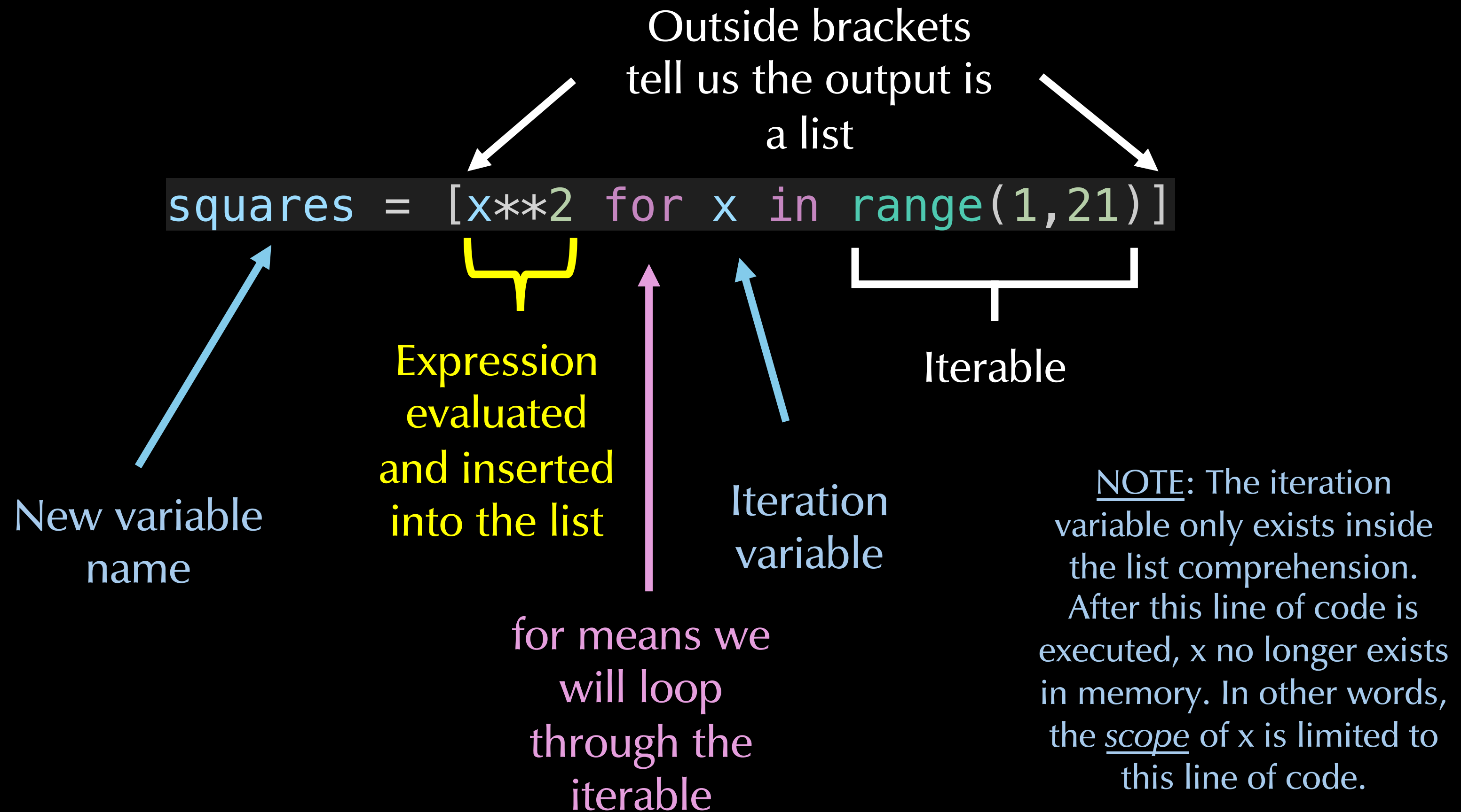
List comprehension (can also be applied to sets and dictionaries)

In Python, comprehensions are powerful constructs for creating lists. They are particularly useful for applying an expression (sometimes a function) to an iterable. They may perform faster and enhance readability.

```
# For loop to create a list with the first 20 squares.
squares = []
for i in range(1,21):
    squares.append(i**2)

# Do the same thing with a list comprehension.
squares = [x**2 for x in range(1,21)] # About 3x faster.
```

Anatomy of a list comprehension



English: For each value of x from 1 to 20, insert x-squared into the list.

Readability and complexity matter too!

Example of list comprehension requiring a function to call another process (print)

```
numbers = range(10)

# To include print statements, you would need to use a helper function
def process_number(num):
     if num % 2 == 0:
          print(f"{num} is even, squared value is {num ** 2}")
          return num ** 2
     else:
          print(f"{num} is odd, cubed value is {num ** 3}")
          return num ** 3

result = [process_number(num) for num in numbers]
print(result)
```

Equivalent for loop

```
numbers = range(10)
result = []

for num in numbers:
     if num % 2 == 0:
          result.append(num ** 2)
          print(f"{num} is even, squared value is {num ** 2}")
     else:
          result.append(num ** 3)
          print(f"{num} is odd, cubed value is {num ** 3}")

print(result)
```

Which one is more readable? Plus, if you execute both, which, if either, is faster?

If statements can also be incorporated

```
oddsquares = [x**2 for x in range(1,21) if x % 2 != 0]
```

In this example, list oddsquares will only get the squares of odd integers up to 20.

```
oddsquaresdict = {x: x**2 for x in range(1,21) if x % 2 != 0}
```

Similar idea, but now we are making a dictionary in which values of x are the keys and x**2 are the values.