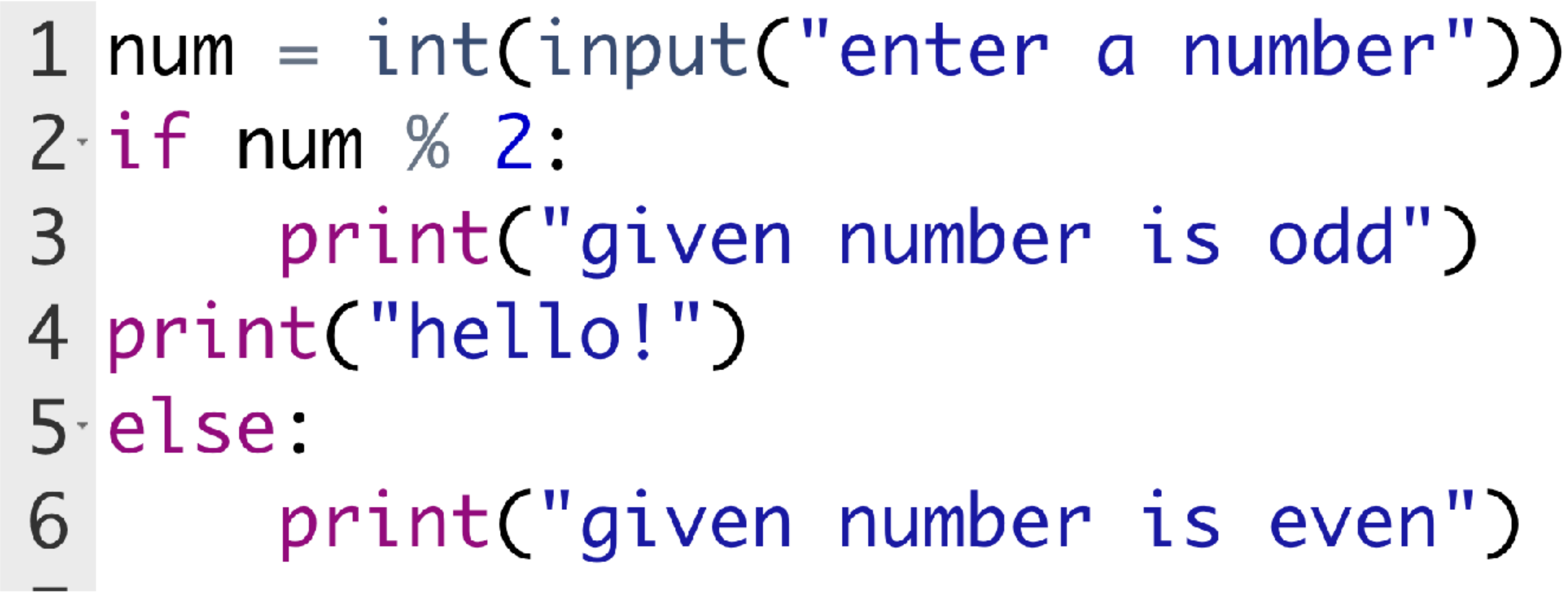
COL1000
Introduction to Programming

Priyanka Golia

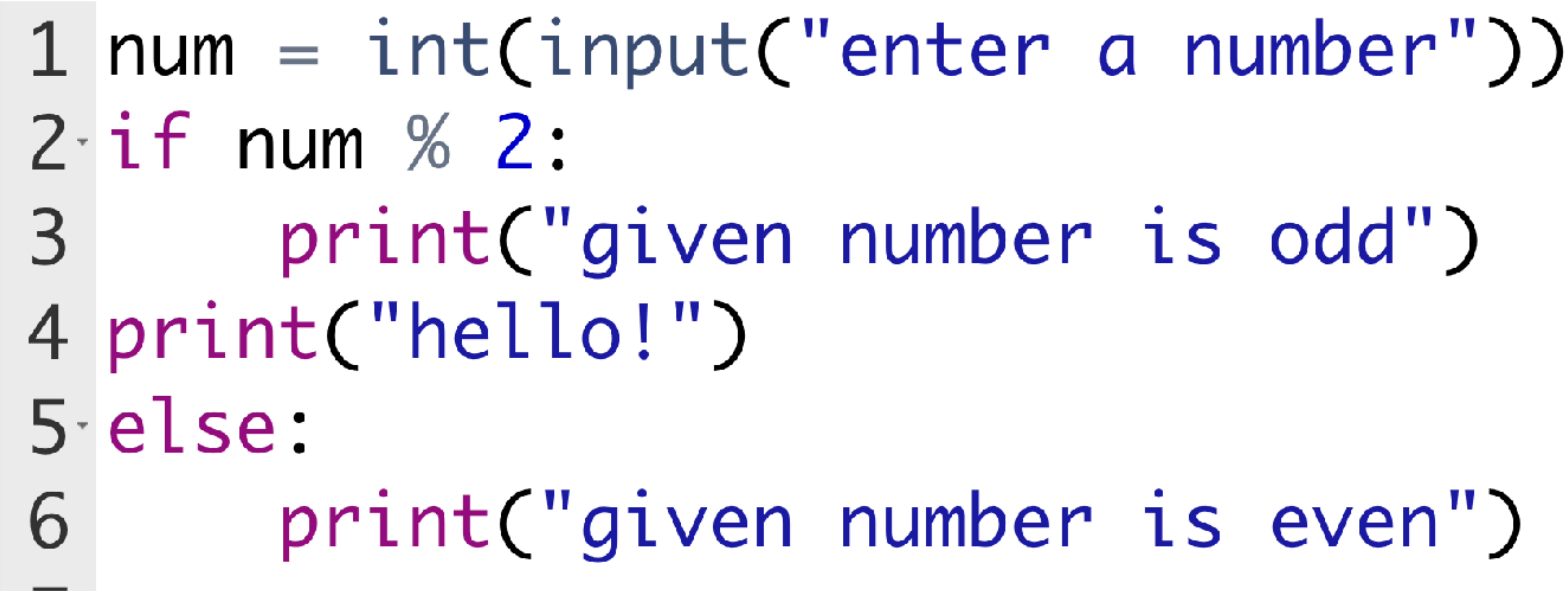
Most (if not all) of the content is borrowed from Prof. Subodh Kumar's slides

If there is an error, what is it?

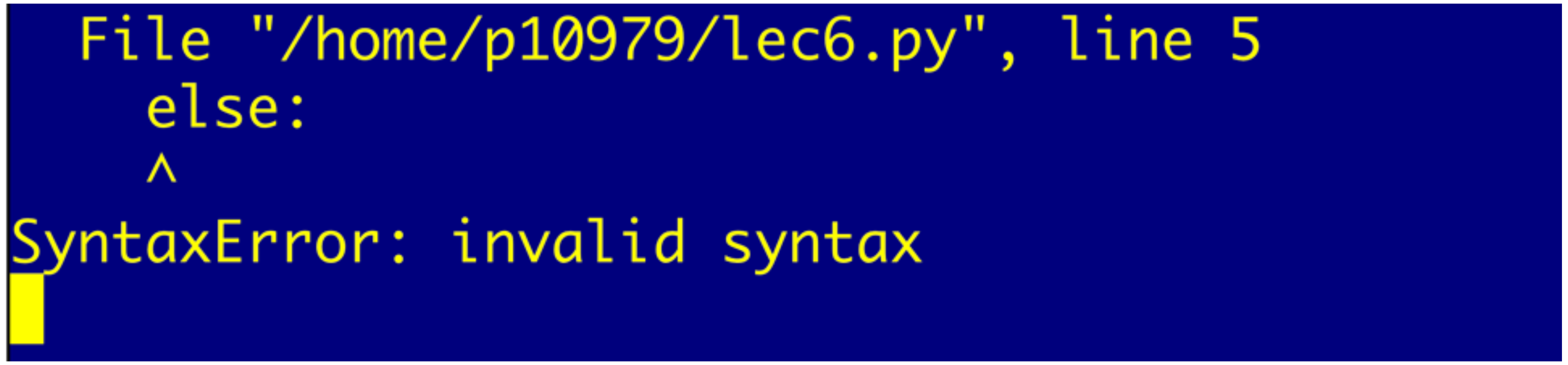


```
num = int(input("enter a number"))
if num % 2:
    print("given number is odd")
print("hello!")
else:
    print("given number is even")
```

If there is an error, what is it?



```
1 num = int(input("enter a number"))
2 if num % 2:
3     print("given number is odd")
4 print("hello!")
5 else:
6     print("given number is even")
```



```
  File "/home/p10979/lec6.py", line 5
    else:
    ^
SyntaxError: invalid syntax
```

Design of Conditions

- Check that the state is as expected (if not: take remedial action)
  - ➡ e.g., Is the input string convertible to an int?
  - ➡ Prevents later errors
- Follow different procedures (commands) for different cases
  - ➡ e.g., if absent: mark = 0.5 * mark
- Find all the different cases that the program needs to handle (at that stage), and divide them in separate branches
  - ➡ Organize into broad cases, then sub-cases within each case, and so on

Design of Conditions

- Check that the state is as expected (if not: take remedial action)
    - ➡ e.g., Is the input string convertible to an int?
    - ➡ Prevents later errors
- Follow different procedures (commands) for different cases
    - ➡ e.g., if absent: mark = 0.5 * mark
- Find all the different cases that the program needs to handle (at that stage), and divide them in separate branches
    - ➡ Organize into broad cases, then sub-cases within each case, and so on

(Top-down design: more later)

Design of Conditions

- A “fork in the execution path”

Not just what to do next, but whether to do so

  1. One of two blocks of statements are executed, based on a “decision” value
  2. Or, the execution of a block of statements could be skipped

- If you execute the program again, a different choice may be made

  3. because the decision value may be different this time

- Fork within a fork *(nesting)* is allowed

Must indent

```
if <boolean object>:
    statements
else:
    statements
```

Remember colons

```
if <boolean object 1>:
    statements1
    if <boolean object 2>:
        statements2
    statements3
else:
    statements4
statements5
```

Design of Conditions

- A "fork in the execution path"

Not just what to do next, but whether to do so

1. One of two blocks of statements are executed, based on a "decision" value
2. Or, the execution of a block of statements could be skipped

- If you execute the program again, a different choice may be made

3. because the decision value may be different this time

- Fork within a fork (*nesting*) is allowed

Must
Indent

```
if <boolean object>:
    statements
else:
    statements
```

Remember
colons

```
if <boolean object 1>:
    statements1
    if <boolean object 2>:
        statements2
    statements3
else:
    statements4
statements5
```

next statement

Example (Design of Conditions)

- *Is given year a leap year*
  - ➡ If year divisible by 4, except years divisible by 100, but not by 400

Divisibility by 4 is an important determinator

Example (Design of Conditions)

- *Is given year a leap year*
  - ➡ If year divisible by 4, except years divisible by 100, but not by 400

Divisibility by 4 is an important determinator

```
# year = …
if year%4: # Not divisible ⇒ always non-leap
    leap = False
else  # divisible by 4, not always leap – depends on other factors
    if year%100: # Not divisible by 100
        leap = True
    else:
        leap = not year%400 # Divisible
```

Example (Design of Conditions)

- *Is given year a leap year*
  - ➡ If year divisible by 4, except years divisible by 100, but not by 400

Divisibility by 4 is an important determinator

```
# year = …
if year%4: # Not divisible ⇒ always non-leap
   leap = False
else  # divisible by 4, not always leap — depends on other factors
    if year%100: # Not divisible by 100
        leap = True
    else:
        leap = not year%400 # Divisible
```

Same as

```
if year % 400:
    leap = False
else:
    leap = True
```

Example (Design of Conditions)

- *Is given year a leap year*
    - ➡ If year divisible by 4, except years divisible by 100, but not by 400

Divisibility by 4 is an important determinator

```
# year = …
if year%4: # Not divisible ⇒ always non-leap
   leap = False
else  # divisible by 4, not always leap — depends on other factors
    if year%100: # Not divisible by 100
        leap = True
    else:
        leap = not year%400 # Divisible
```

Same as

```
if year % 400:
    leap = False
else:
    leap = True
```

Same as

```
leap = not year%4 and year%100 or not year%400
```

Example (Design of Conditions)

- *Is given year a leap year*
  - ➡ If year divisible by 4, except years divisible by 100, but not by 400

Divisibility by 4 is an important determinator

```
# year = …
if year%4: # Not divisible ⇒ always non-leap
   leap = False
else  # divisible by 4, not always leap — depends on other factors
    if year%100: # Not divisible by 100
        leap = True
    else:
        leap = not year%400 # Divisible
```

Same as

```
if year % 400:
    leap = False
else:
    leap = True
```

Same as

```
leap = not year%4 and year%100 or not year%400
```

No precedence override is needed here 👆 % ⧁ not ⧁ and ⧁ or

To test if a given number is a prime number or not.

```
if number < 2:
   print(number, 'is not prime')
else: # number is >= 2
   if number == 2:
      print(number, 'is prime')
   else: # number is > 2
      if number % 2 == 0: # it’s even
         print(number, 'is not prime')
      else: # Neither 2 nor even
         if number == 3:
            print(number, 'is prime')
         else:
            if number % 3 == 0: # divisible by 3
               print(number, 'is not prime')
            else: # Odd, >3, Not divisible by 3
               if number == 5:
                  print(number, 'is prime')
               else:
                  if number % 5 == 0: # divisible by 5
                     print(number, 'is not prime')
                  else:
                     if number == 7:
                        print(number, 'is prime')
                     else:
                        if number % 7 == 0: # divisible by 7
                           print(number, 'is not prime')
                        else:
                           print(“C'mon! Stop already.")
```

To test if a given number is a prime number or not.

```
if number < 2:
   print(number, 'is not prime')
else: # number is >= 2
   if number == 2:
      print(number, 'is prime')
   else: # number is > 2
      if number % 2 == 0: # it’s even
         print(number, 'is not prime')
      else: # Neither 2 nor even
         if number == 3:
            print(number, 'is prime')
         else:
            if number % 3 == 0: # divisibl
               print(number, 'is not prime
            else: # Odd, >3, Not divisible
               if number == 5:
                  print(number, 'is prime'
               else:
                  if number % 5 == 0: # di
                     print(number, 'is not
                  else:
                     if number == 7:
                        print(number, 'is
                     else:
                        if number % 7 == 0
                           print(number, '
                        else:
                           print(“C'mon! S
```

```
if number < 2:
   print(number, 'is not prime')
elif number == 2:
   print(number, 'is prime')
elif number % 2 == 0: # it’s even
   print(number, 'is not prime')
elif number == 3:
   print(number, 'is prime')
elif number % 3 == 0: # divisible by
3
   print(number, 'is not prime')
elif number == 5:
   print(number, 'is prime')
elif number % 5 == 0: # divisible by
5
   print(number, 'is not prime')
elif number == 7:
   print(number, 'is prime')
elif number % 7 == 0: # divisible by
7
   print(number, 'is not prime')
elif number == 11:
   print(number, 'is prime')
else: print("C'mon! Enough
already.")
```

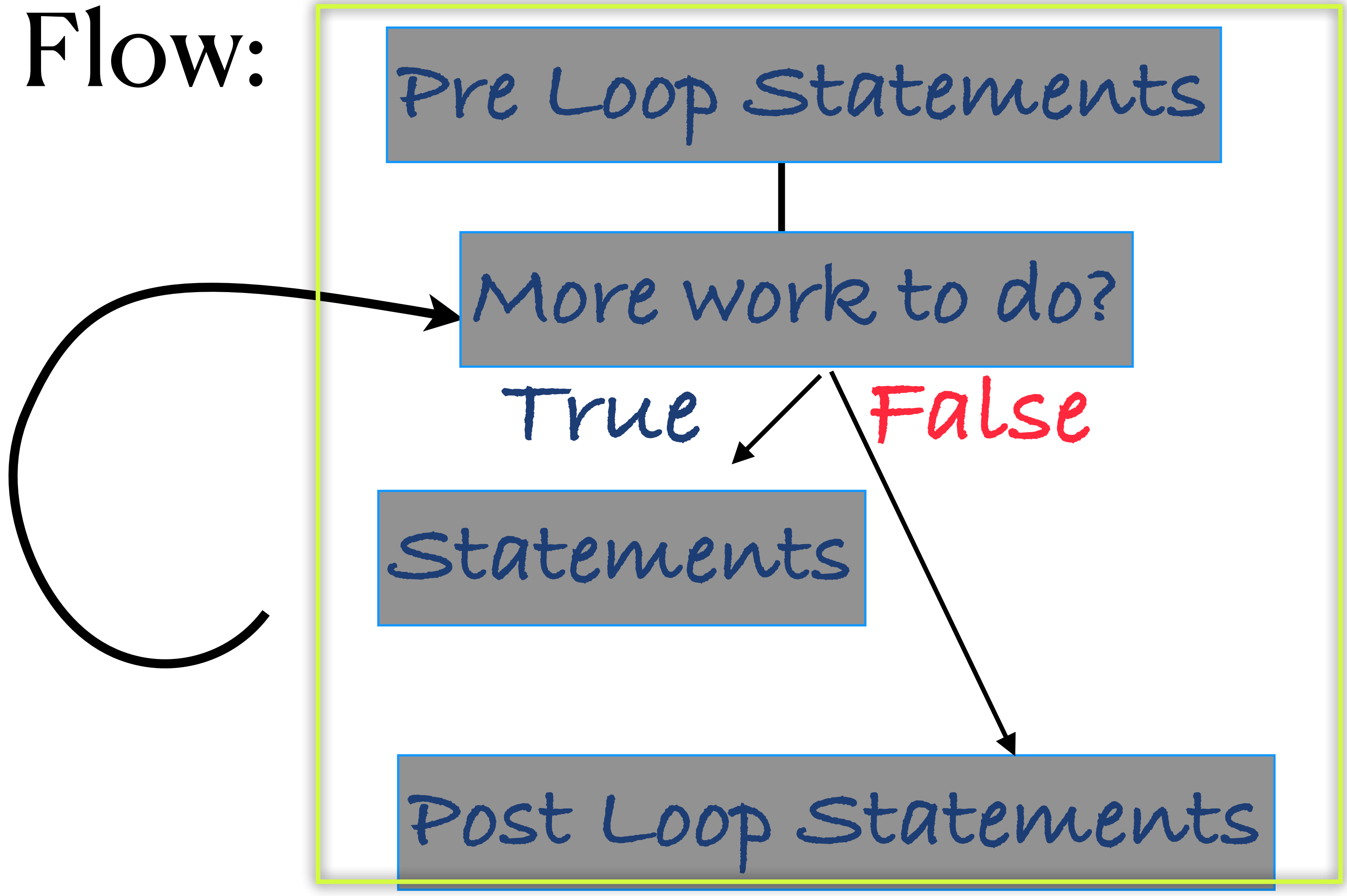
```
if number < 2:
   print(number, 'is not prime')
elif number == 2:
   print(number, 'is prime')
elif number % 2 == 0: # it’s even
   print(number, 'is not prime')
elif number == 3:
   print(number, 'is prime')
elif number % 3 == 0: # divisible by
3
   print(number, 'is not prime')
elif number == 5:
   print(number, 'is prime')
elif number % 5 == 0: # divisible by
5
   print(number, 'is not prime')
elif number == 7:
   print(number, 'is prime')
elif number % 7 == 0: # divisible by
7
   print(number, 'is not prime')
elif number == 11:
   print(number, 'is prime')
else: print("C'mon! Enough
already.")
```

Keep going

```
if number < 2:
   print(number, 'is not prime')
elif number == 2:
   print(number, 'is prime')
elif number % 2 == 0: # it’s even
   print(number, 'is not prime')
elif number == 3:
   print(number, 'is prime')
elif number % 3 == 0: # divisible by 3
   print(number, 'is not prime')
elif number == 5:
   print(number, 'is prime')
elif number % 5 == 0: # divisible by 5
   print(number, 'is not prime')
elif number == 7:
   print(number, 'is prime')
elif number % 7 == 0: # divisible by 7
   print(number, 'is not prime')
elif number == 11:
   print(number, 'is prime')
else: print("C'mon! Enough already.")
```

Keep going

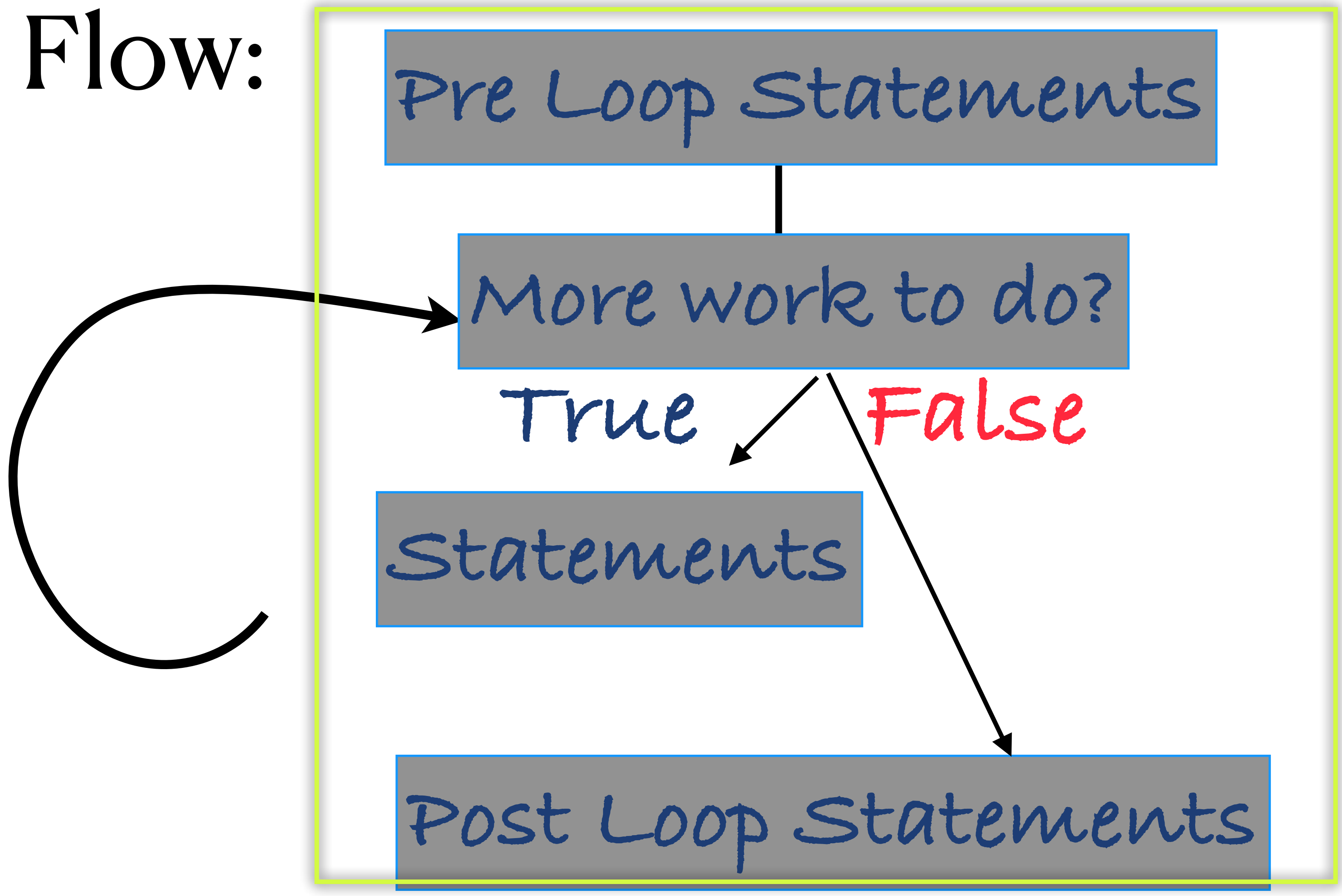
Control Flow:



```
if number < 2:
   print(number, 'is not prime')
elif number == 2:
   print(number, 'is prime')
elif number % 2 == 0: # it’s even
   print(number, 'is not prime')
elif number == 3:
   print(number, 'is prime')
elif number % 3 == 0: # divisible by
3
   print(number, 'is not prime')
elif number == 5:
   print(number, 'is prime')
elif number % 5 == 0: # divisible by
5
   print(number, 'is not prime')
elif number == 7:
   print(number, 'is prime')
elif number % 7 == 0: # divisible by
7
   print(number, 'is not prime')
elif number == 11:
   print(number, 'is prime')
else: print("C'mon! Enough
already.")
```

Keep going

Control Flow:



We call this a “loop”

Do something Similar &
Keep going until “done”

“Range”

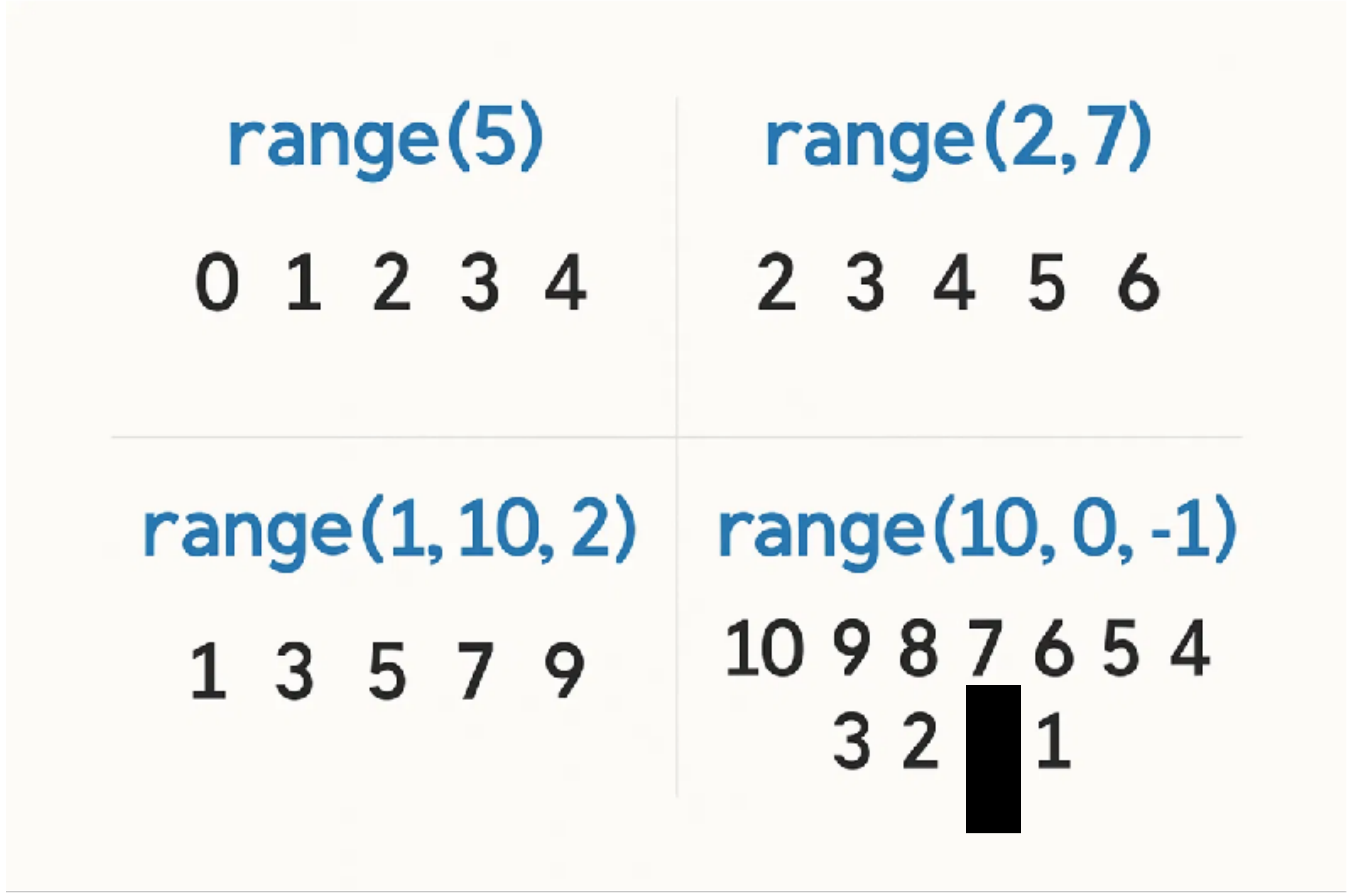
Range creates an object of type “range”. It is a sequence of numbers, commonly used with for loops to repeat an action a specific number of times.

```
# range(stop) - starts at 0
for i in range(5):
 print(i)       # 0, 1, 2, 3, 4

# range(start, stop) - custom start
for i in range(2, 7):
 print(i)        # 2, 3, 4, 5, 6

# range(start, stop, step)
for i in  range(1, 10, 2):
 print(i)     # 1, 3, 5, 7, 9
```

Stop value is exclusive (not included)



range(5)

0 1 2 3 4

range(2,7)

2 3 4 5 6

range(1,10,2)

1 3 5 7 9

range(10,0,-1)

10 9 8 7 6 5 4
3 2 [redacted] 1

Step can be negative for counting down

Loops

- Example: *Sum positive integers upto n*
- The “for” construct

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

```
for i in range(n+1):
```

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

```
for i in range(n+1):
```

☞ Recall:

This Creates an object of the “range” type

By default, a single number denotes the end of the range

(This end is non-inclusive)

The range begins at 0 by default

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

```
for i in range(1, n+1):
```

☜ Recall:

This Creates an object of the “range” type

By default, a single number denotes the end of the range

(This end is non-inclusive)

The range begins at 0 by default

Loops

- Example: *Sum positive integers upto n*

- The "for" construct

i is renamed to all values in the range, one after another

```
for i in range(1, n+1):
```

☞ Recall:

This Creates an object of the "range" type

By default, a single number denotes the end of the range

(This end is non-inclusive)

The range begins at 0 by default

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

i is renamed to all values in the range, one after another

```
for i in range(1, n+1):
```

add i… but to what?

☞ Recall:

This Creates an object of the “range” type

By default, a single number denotes the end of the range

(This end is non-inclusive)

The range begins at 0 by default

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

i is renamed to all values in the range, one after another

```
for i in range(1, n+1):
```

add i… but to what?

Need to “remember”: use variable ☞

☜ Recall:

This Creates an object of the “range” type

By default, a single number denotes the end of the range

(This end is non-inclusive)

The range begins at 0 by default

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

i is renamed to all values in the range, one after another

```
for i in range(1, n+1):
    sum = sum + i
```

Need to “remember”: use variable

☜ Recall:

This Creates an object of the “range” type

By default, a single number denotes the end of the range (This end is non-inclusive)

The range begins at 0 by default

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

i is renamed to all values in the range, one after another

Colon, just like with “if”

Indent

```
for i in range(1, n+1):
    sum = sum + i
```

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

i is renamed to all values in the range, one after another

Colon, just like with “if”

Indent

```
for i in range(1, n+1):
    sum = sum + i
print('Sum is', sum)
```

Statement after the loop statement

Loops

- Example: *Sum positive integers upto n*

- The "for" construct

i is renamed to all values in the range, one after another

Colon, just like with "if"

Indent

```
sum = 0
for i in range(1, n+1):
    sum = sum + i
print('Sum is', sum)
```

Statement after the loop statement

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

i is renamed to all values in the range, one after another

Colon, just like with “if”

Indent

```
sum = 0
for i in range(1, n+1):
    sum = sum + i
print('Sum is', sum)
```

Statement after the loop statement

- Loops have three parts:

Before Loop

  - ➡ Initialization; Progress; Termination

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

i is renamed to all values in the range, one after another

Colon, just like with “if”

```
sum = 0
for i in range(1, n+1):
    sum = sum + i
print('Sum is', sum)
```

Indent

Statement after the loop statement

- Loops have three parts:

Before Loop

pattern

➡ Initialization; Progress; Termination

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

i is renamed to all values in the range, one after another

```
sum = 0
for i in range(1, n+1):
    sum = sum + i
print('Sum is', sum)
```

Colon, just like with “if”

Indent

Statement after the loop statement

- Loops have three parts:

Before Loop pattern Condition

  ➡ Initialization; Progress; Termination

Loops

- Example: *Sum positive integers upto n*

- The “for” construct

i is renamed to all values in the range, one after another

Colon, just like with “if”

Indent

```
sum = 0
for i in range(1, n+1):
    sum = sum + i
print('Sum is', sum)
```

Statement after the loop statement

- Loops have three parts:

Before Loop   pattern   Condition

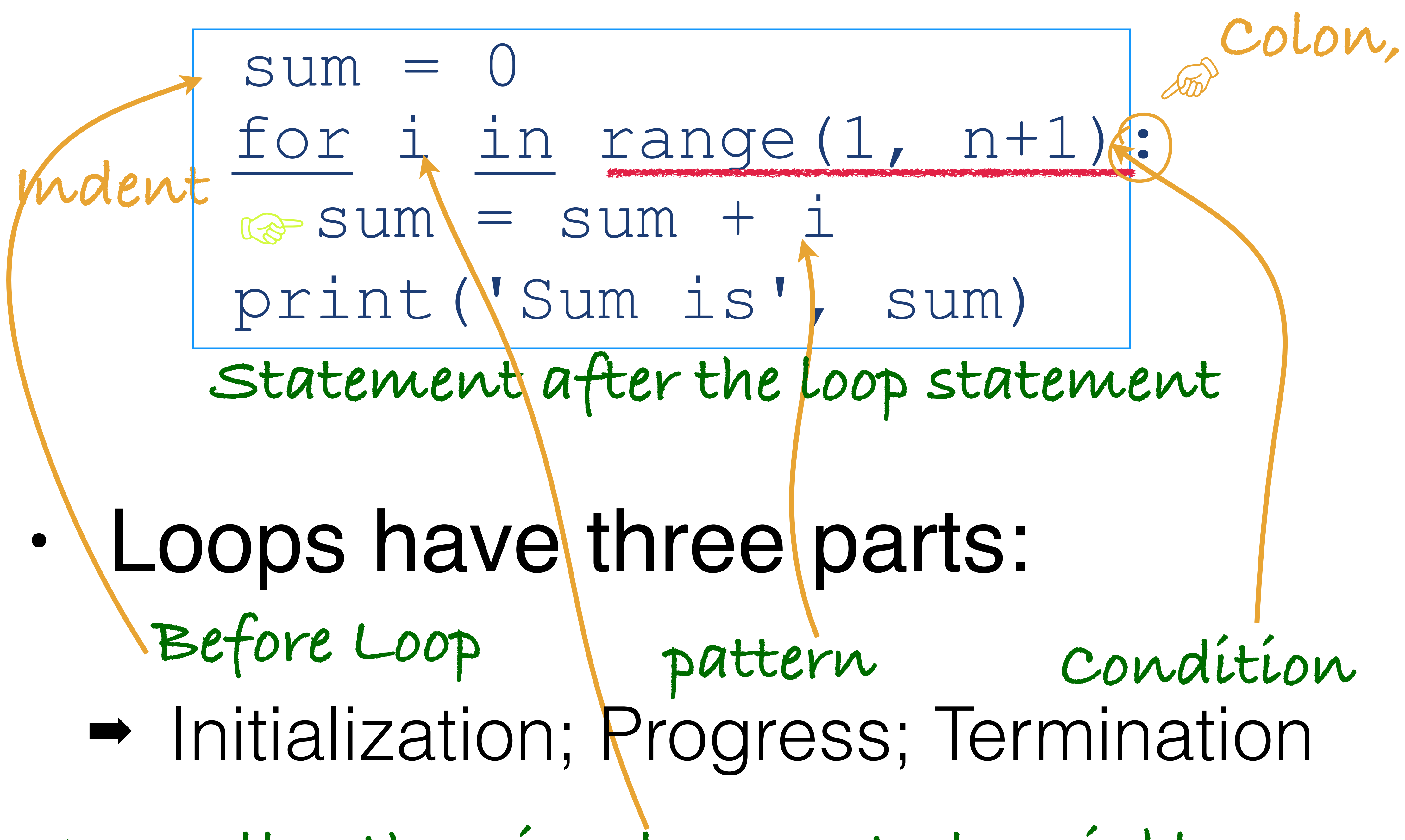
➡ Initialization; Progress; Termination

Generally, there is a loop control variable
(that drives the pattern and termination)

Loops

- Example: *Sum positive integers upto n*
- The “for” construct

i is renamed to all values in the range, one after another



Colon, just like with “if”

```
sum = 0
for i in range(1, n+1):
    sum = sum + i
print('Sum is', sum)
```

Statement after the loop statement

- Loops have three parts:

  Before Loop   pattern   Condition

  ➡ Initialization; Progress; Termination

Generally, there is a loop control variable (that drives the pattern and termination)

The most important aspect of loop design: How each individual iteration moves the ball closer to the goal: how it modifies the partial solution from the previous iteration to provide to the next iteration