# Lecture 6

## Iteration

# `if-else` Condition

```
if <expression>:
      <body>
```

- body
  - Body will be evaluated if predicate `<expression>` is `True`

# `if-else` Condition

```
total = 0

if total < 100:
    total = total + 1
```

**The body will only execute once!**

# `while` loop

```
while <expression>:
      <body>
```

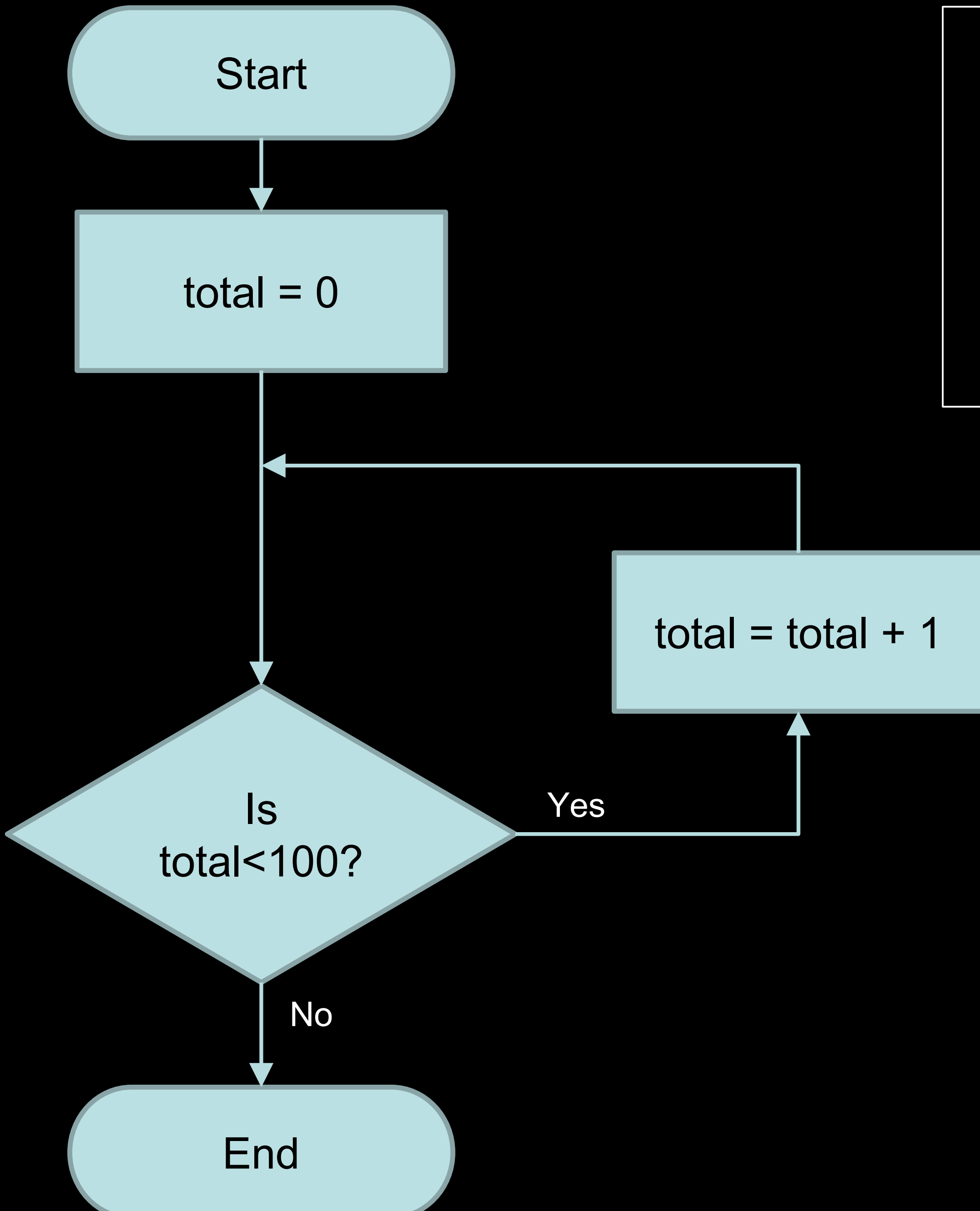
- expression
  - Predicate to stay within loop
- body
  - The body will be evaluated if predicate <expression> is `True`

# `while` Loop

```
total = 0

while total < 100:
      total = total + 1
```

**The body will execute until the <expression> is False.**




# `while` Loop

```
total = 0

while total < 100:
    total = total + 1
```

# Recap: Factorial

```
factorial(6)
6 * factorial(5)
6 * 5 * factorial(4)
6 * 5 * 4 * factorial(3)
6 * 5 * 4 * 3 * factorial(2)
6 * 5 * 4 * 3 * 2 * factorial(1)
6 * 5 * 4 * 3 * 2 * 1
6 * 5 * 4 * 3 * 2
6 * 5 * 4 * 6
6 * 5 * 24
6 * 120
720
```

Recursive Process

# Factorial using while-loop

- Start with 1, multiply by 2, multiply by 3, …
- Factorial rule:

  Initiatialise : product $\leftarrow$ 1, counter $\leftarrow$ 1

  product $\leftarrow$ product * counter

  counter $\leftarrow$ counter + 1

```
def factorial(n):
    product, counter = 1, 1
    while counter <= n:
        product = product * counter
        counter = counter + 1
    return product
```

# Iterative process

```
def factorial(n):
    product, counter = 1, 1
    while counter <= n:
        product = (product *
                   counter)
        counter = counter + 1
    return product

factorial(6)
```

| Product | Counter |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 2 | 3 |
| 6 | 4 |
| 24 | 5 |
| 120 | 6 |
| 720 | 7 |

**counter > n**

**return product = 720**

# Alternatively

- Start with n, multiply by (n-1), multiply by (n-2), … multiply by 3, 2 and then 1.
- Factorial rule: Initiatialise : product ← 1, counter ← n

  product ← product * counter

  counter ← counter - 1

```
def factorial(n):
    product, counter = 1, n
    while counter > 0:
        product = product * counter
        counter = counter - 1
    return product
```

# `for` Loop (1)

```
for ele in seq :

    <body>
```

- **<body>**
  **The body will be evaluated once for each element inside the sequence.**

- **<seq>**
  **Some examples of sequence (seq) are** `String, List, Tuple, Dictionary ...`

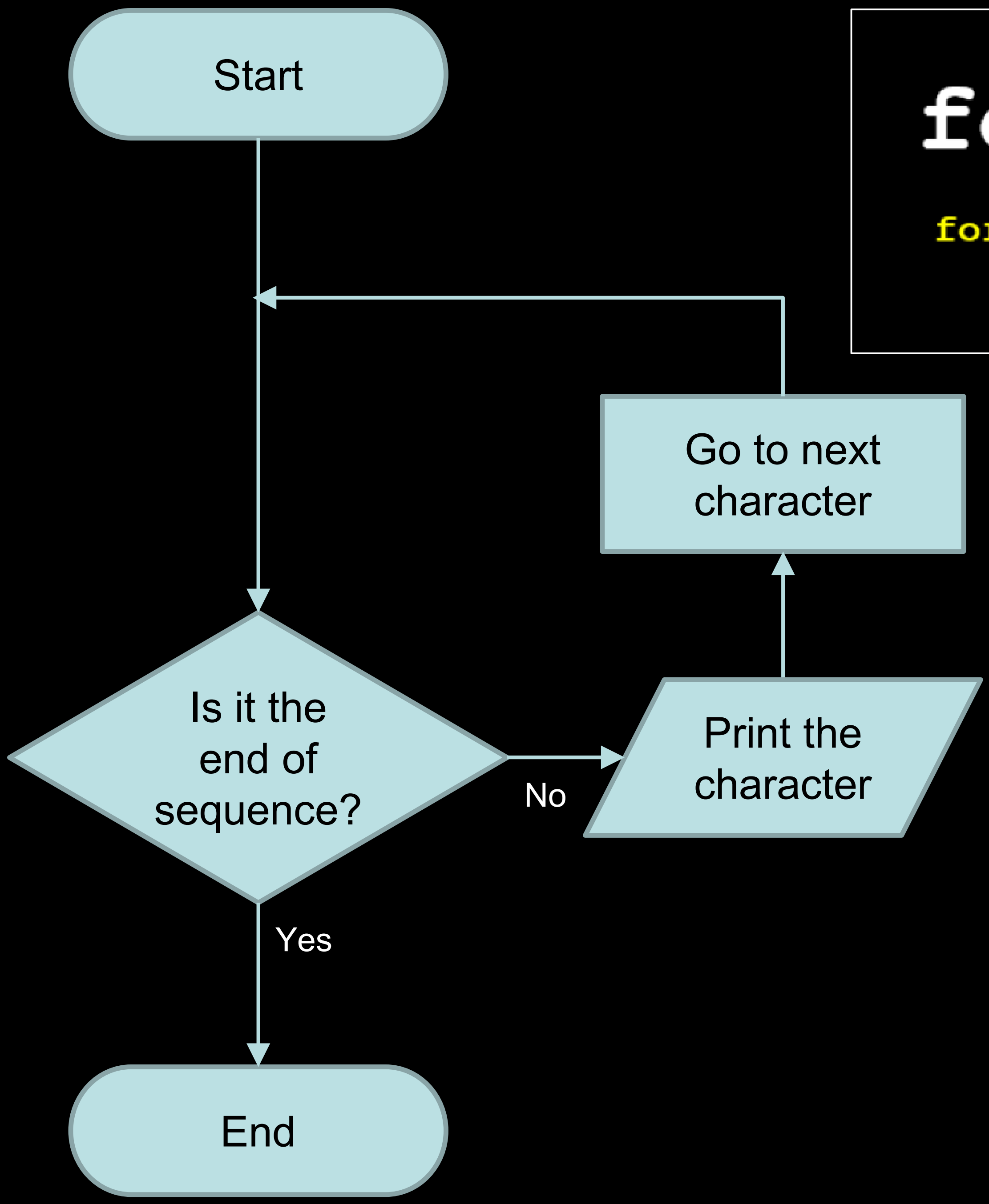
# `for` Loop (1)

```
for char in 'Singapore':
    print(char)

>>> S
    i
    n
    g
    a
    p
    o
    r
    e
```

# `for` Loop (1)

```
for char in 'Singapore':
    print(char)
```



**while - loop:**

- **The process in the body repeated until a certain condition is met.**

**for - loop:**

- **The process in the body repeated for a fixed number of times.**

# `for` Loop (2)

```
for i in range(start,stop,step):

    <body>
```

- <body>

    The body be evaluated once for each of `i`, from `start` to `(stop-1)` incrementally in intervals of `step`.

# `for` loop (2)

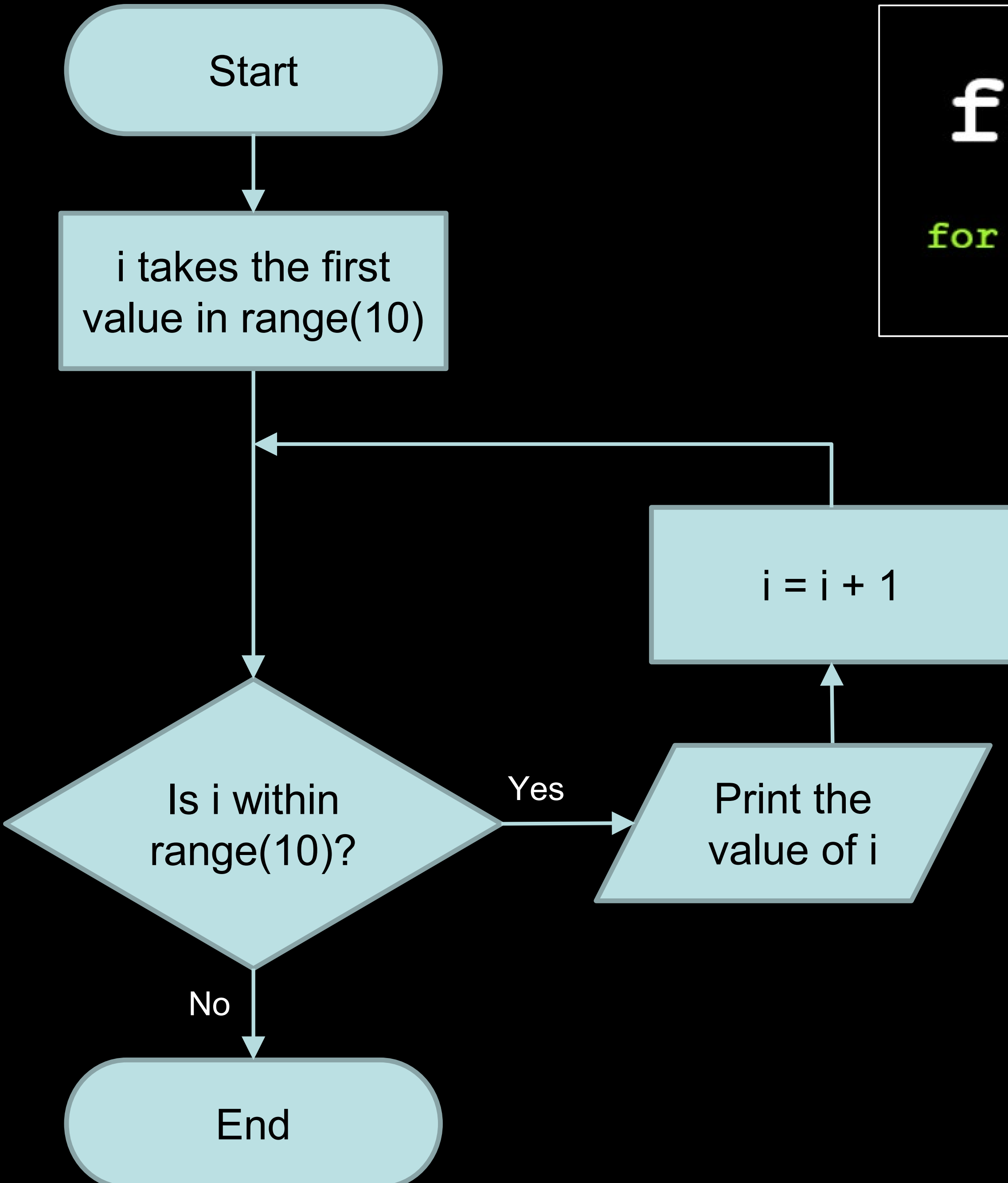
```
for i in range(10):
    print(i)

>>>  0
     1
     2
     3
     .
     .
     .
     8
     9
```

Same as range(0, 10, 1)

# for loop (2)

```python
for i in range(10):
    print(i)
```



# `for` loop (2)

```
for i in range(3,10):
     print(i)

>>>  3
     4
     5
     .
     .
     .
     8
     9
```

Same as
range(3, 10, 1)

# `for` loop (2)

```
for i in range(3,10,2):
     print(i)

>>>  3
     5
     7
     9
```

# for Loop (2)

```
string = 'Singapore'
for i in range(len(string)):
    print(string[i])

>>> S ← string[0]
    i ← string[1]
    n ← string[2]
    g ← string[3]
    a ← string[4]
    p ← string[5]
    o ← string[6]
    r ← string[7]
    e ← string[8]
```

len(string) = 9

# `for` Loop (2)

```
string = 'Singapore'
for i in range(0,len(string),2):
    print(string[i])

>>> S ← string[0]
    n ← string[2]
    a ← string[4]
    o ← string[6]
    e ← string[8]
```

len(string) = 9

# Factorial using for-loop

- Start with 1, multiply by 2, multiply by 3, …
- Factorial rule:

  product ← product * counter

  counter ← counter + 1

**Up to n, does not including n+1**

```
def factorial(n):
    product = 1
    for counter in range(1,n+1):
        product = product * counter
    return product
```

# Iterative process

```
def factorial(n):
    product = 1
    for i in range(1,n+1):
        product = product * i
    return product

factorial(6)
```

| i | Product |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 6 |
| 4 | 24 |
| 5 | 120 |
| 6 | 720 |

**return product = 720**

# Iterative process

`factorial(6)`

| Product | Counter |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 2 | 3 |
| 6 | 4 |
| 24 | 5 |
| 120 | 6 |
| 720 | 7 |

- Number of steps: linearly proportional to *n*
- Space required: **constant**
- No deferred operations.
- All relevant information contained in variables.

# break & continue

```
for j in range(10):
    print(j)
    if j == 3:
        break
```

0
1
2
3

**Break out of loop**

```
for j in range(10):
    if j % 2 == 0:
        continue
    print(j)
```

1
3
5
7
9

**Skip current iteration**

# Recursion

# vs

# Iteration

**Recursive process:**

- occurs when there are deferred operations
- recursive process is straightforward (and more elegant)

**Iterative process:**

- does not have deferred operations
- (usually) more efficient

# Looking forward…

Different ways of performing a computation (algorithms) can consume **dramatically** different amounts of resources.

# Designing programmes

- Must work
- Efficiency
    - Time taken (time complexity)
    - Amount of memory used (space complexity)

# Measuring Time

- In terms of basic steps executed
- Assume that
  - steps are executed in sequence
  - Each step is an operation that takes constant time

```python
def f(n):
    answer = 1
    if n == 0:
        return  answer
    else:
        while n>1:
            answer *= n
            n -= 1
    return answer
```

# But there is a problem!

- Depending on the value we choose, the complexity might change

```
def linearSearch(L, x):
    for e in L:
        if e == x:
            return True
    return False
```

**Best case: x is at the start of the list**

**Worst case: x cannot be found**

**Average case: x is in the middle**

**We will use the Worst-Case Scenario**

# Example 1

- What is the order of growth of the following function?

```
def g(n):
    return n*2
```

Big-O notation

**O(1)**

1 step, no matter the value of n

# Example 2

- What is the order of growth of the following function?

```
def fact(n):
    answer = 1
    while n>1:
        answer*=n
        n-=1
    return answer
```

**1 + 2*(n-1) + 1**

# O(n)

# Example 3

- What is the order of growth of the following function?

```
def f(x):
    for i in range(1000):
        ans=i
    for i in range(x):
        ans += 1
    for i in range(x):
        for j in range(x):
            ans += 1
```

**O(n^2)**

**1000 + x + x^2**

# Rule to calculate Order of Growth

1. Ignore the constants
   - They become irrelevant when n becomes very big
2. Ignore the constant multiples
   - When n is big, it doesn't matter, e.g. if it takes 3000 hours or 6000 hours
3. Take the term with the largest growth rate
   - This term contributes the most to the growth rate