CL08 – Variables & while loops

Announcements

Re: Quiz 01:

- Great job! We'll have these returned back to you soon.
- Once grades are published, visit Office Hours to go over quiz questions you missed – we *want* to help you!

LS08: Variables due tonight at 11:59pm

LS09: While Loops due tonight at 11:59pm

EX02: Wordle – due Sunday at 11:59pm

- Uses concepts from last unit and today

Warm-Up: Discuss these questions with a neighbor, then diagram how you believe this works:

```
1    def f(x: int) -> int:
2        y: int
3        y = x * 2
4        return y
5
6
7    print(f(3))
```

Questions to discuss with a neighbor:

What does line 2 remind you of?

What does line 3 remind you of?

```python
def f(x: int) -> int:
    y: int
    y = x * 2
    return y


print(f(3))
```

```python
def pizza_price(size: int) -> float:
    """Calculate the price of a pizza."""
    price: float = 10.0

    if size >= 16:
        price = 20.0

    return price


print(pizza_price(size=16))
```

Key Variable Terminology

Variable Declaration / Definition - Associates a name/identifier with a data type, and a space in the current frame

```
<name>: <type>
```

Examples:

```
students: int
```

```
message: str
```

Key Variable Terminology

Variable Declaration / Definition

```
<name>: <type>
```

- Associates a name/identifier with a data type, and a space in the current frame

Variable Assignment

```
students = 300
```

- Binds a new value to a variable name in memory

Key Variable Terminology

Variable Declaration / Definition

```
<name>: <type>
```

- Associates a name/identifier with a data type, and a space in the current frame

Variable Assignment

```
students = 300
```

- Binds a new value to a variable name in memory

Variable Initialization

- First time a variable is assigned

Key Variable Terminology

Variable Declaration / Definition

```
<name>: <type>
```

- Associates a name/identifier with a data type, and a space in the current frame

Variable Assignment

```
students = 300
```

- Binds a new value to a variable name in memory

Variable Initialization

- First time a variable is assigned

Variable Access

- “Reading” or using a variable name in an expression

Left-hand vs. Right-hand Side of Assignment

Each side of the assignment operator (=) plays a distinct role in variable assignment!

Identify key concepts, then trace the program in a diagram!

Identify: Declaration, Initialization vs. Assignment, Access

```
1   def pizza_price(size: int, toppings: int) -> float:
2       """Calculate the price of a pizza with toppings."""
3       price: float = 10.0
4
5       if size >= 16:
6           price = 20.0
7
8       price = price + toppings * 0.75
9
10      return price
11
12
13  print(pizza_price(size=14, toppings=2))
```

```python
def pizza_price(size: int, toppings: int) -> float:
    """Calculate the price of a pizza with toppings."""
    price: float = 10.0

    if size >= 16:
        price = 20.0

    price = price + toppings * 0.75

    return price


print(pizza_price(size=14, toppings=2))
```

Common Variable Errors

UnboundLocalError – Occurs when attempting to access a variable that is declared in a function but not yet initialized

NameError – Occurs when attempting to access a variable that has not been declared. Commonly from typos or renaming a variable and not updating all accesses

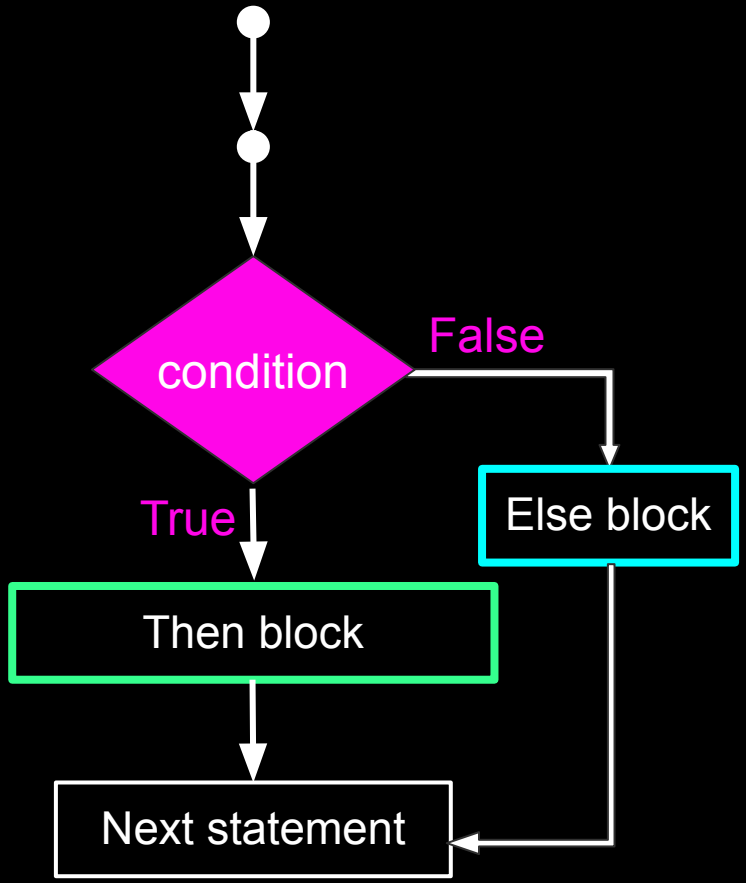
Why variables?

One reason: to store the results of function calls for later use!

```
1   def pizza_price(size: int, toppings: int) -> float:
2       """Calculate price of pizza with toppings."""
3       price: float = 10.0
4
5       if size >= 16:
6           price = 20.0
7
8       price = price + toppings * 0.75
9
10      return price
11
12
13  total_price: float = pizza_price(size=14, toppings=2)
14  print(total_price)
```

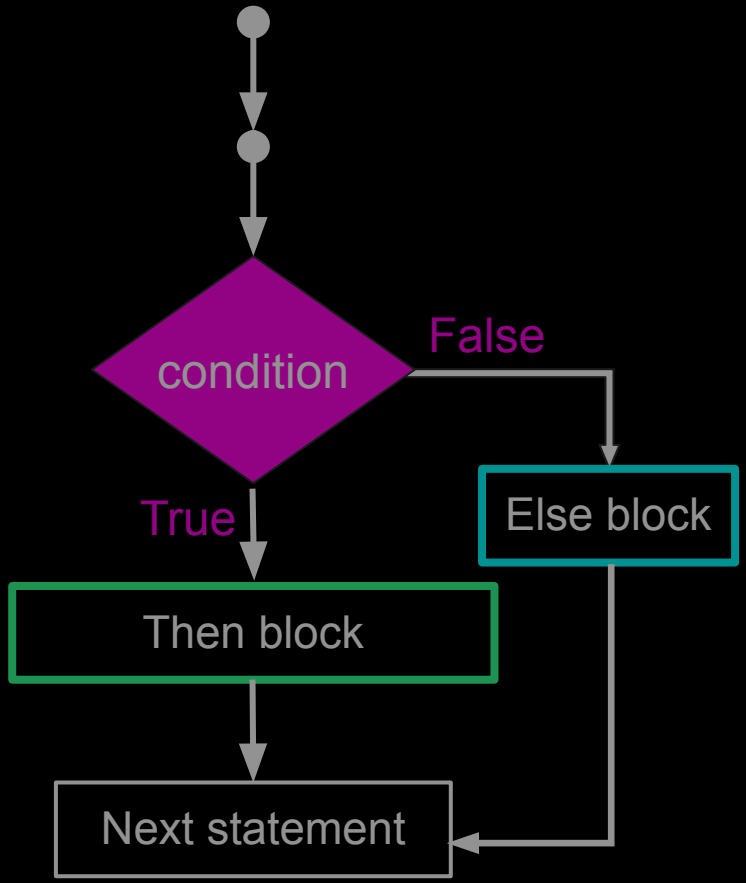
```python
def pizza_price(size: int, toppings: int) -> float:
    """Calculate price of pizza with toppings."""
    price: float = 10.0

    if size >= 16:
        price = 20.0

    price = price + toppings * 0.75

    return price


total_price: float = pizza_price(size=14, toppings=2)
print(total_price)
```

Recall: if-then-else / *Conditional* Statements

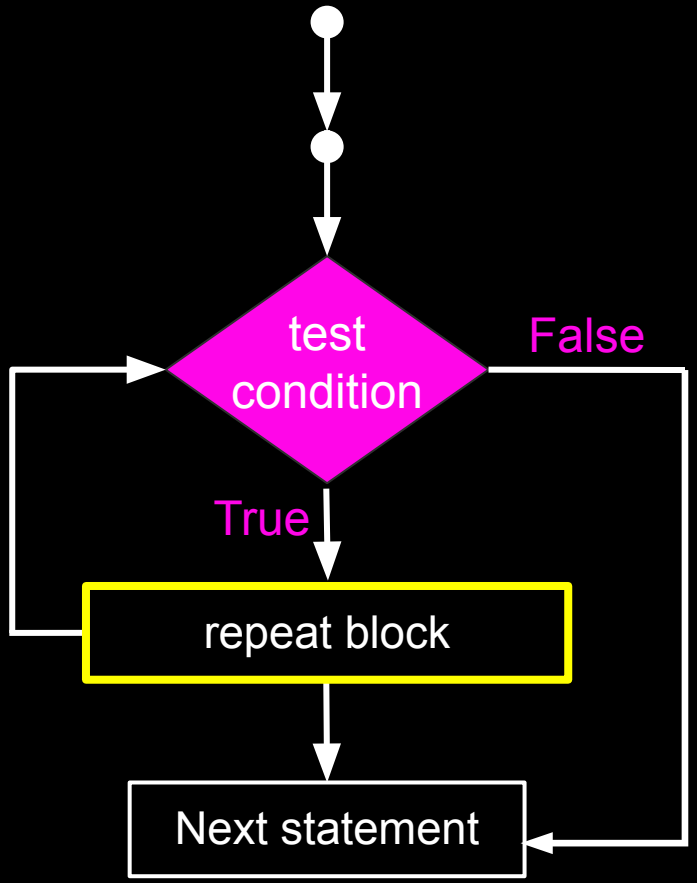



```
if <condition>:
    <then, execute these statements>
else:
    <execute these statements>
<rest of program>
```

`if-then-else` Statements

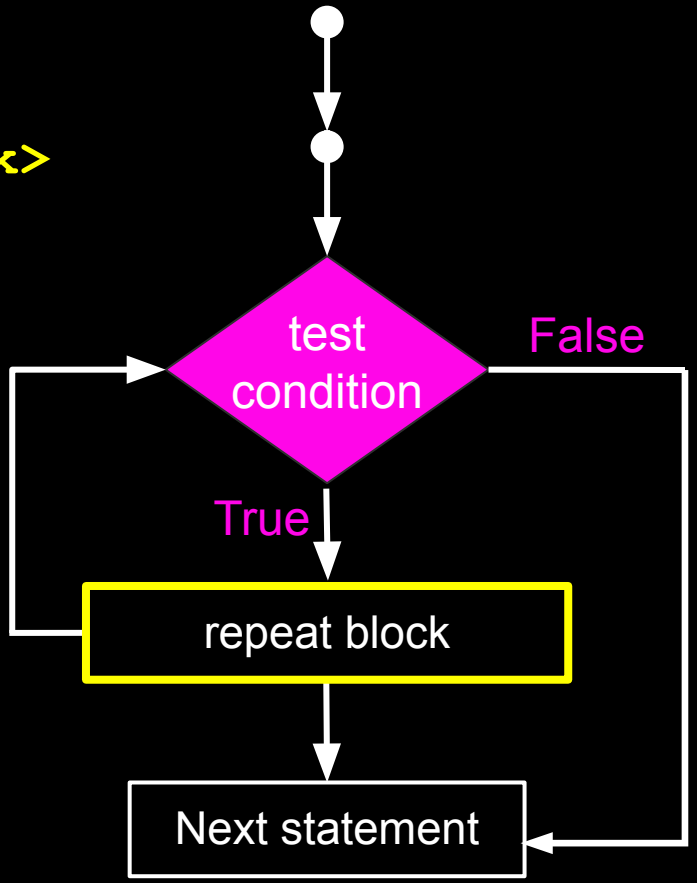


`while` Loop Statements



`while` Loop Statements

```
while <condition>:
    <execute indented repeat block>
<rest of program>
```

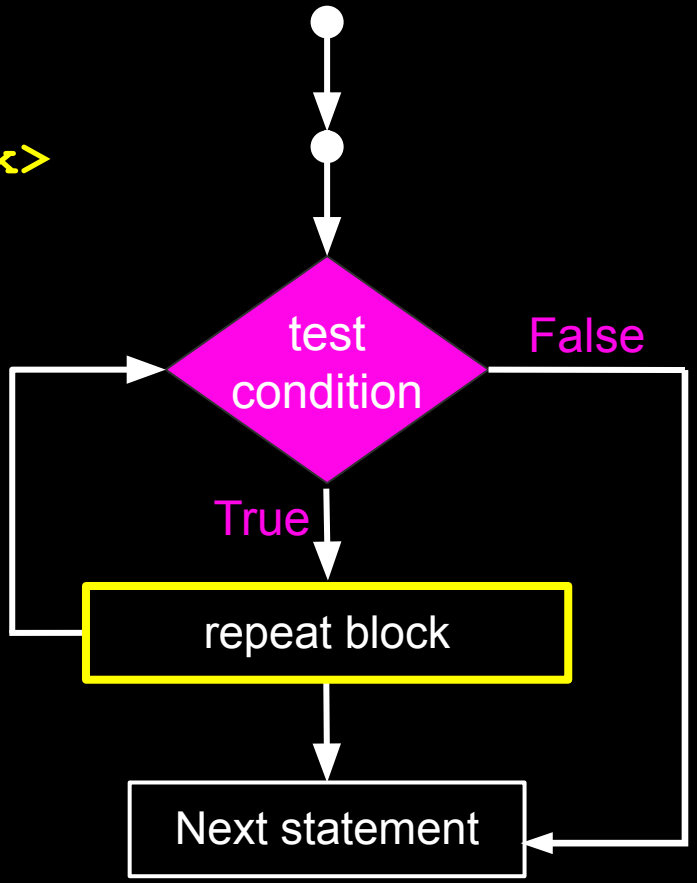


`while` Loop Statements

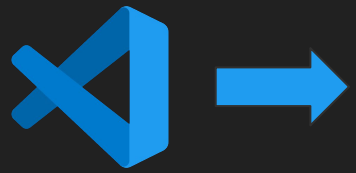
```
while <condition>:

    <execute indented repeat block>

<rest of program>
```

When we reach a while loop statement in code…

- While the **condition** evaluates to **True**:
  - Execute the **repeat block**
  - Jump back up to the test if the **condition** is still True. This process will repeat (“iterate”) until the condition is False. In which case…
- When the **condition** evaluates to **False**:
  - *Skip past* the **repeat block** and continue on to the next line of code at the same level of indentation as the `while` keyword



Let's try writing a function, **`count_to_n`**, that will print values from 0 to n using a **`while`** loop!



Requirements:

Name: `count_to_n`
Parameter: `n`, an int
Return type: `None`

We'll need:

- Local variable (to keep track of the count)
- **`while`** loop (to iterate through each value of count, from 0 to `n`)

Output:

```
Count is: 0
Count is: 1
Count is: 2
Count is: 3
Count is: 4
```

```python
def count_to_n(n: int) -> None:
    count: int = 0
    while count <= n:
        print(f"Count is: {count}")
        count = count + 1


count_to_n(n=4)
```

A common problem: the dreaded *infinite loop*

If a condition in a **`while`** loop never becomes False, the loop will continue indefinitely.

To prevent this:

- Ensure that your loop's condition will eventually be False!

```
1  def count_to_n(n: int) -> None:
2      count: int = 0
3      while count <= n:
4          print(f"Count is: {count}")
5          count = count + 1
6
7
8  count_to_n(n=4)
```

A common problem: the dreaded *infinite loop*

If a condition in a **`while`** loop never becomes False, the loop will continue indefinitely.

To prevent this:

- Ensure that your loop's condition will eventually be False!

```
1  def count_to_n(n: int) -> None:
2      count: int = 0
3      while count <= n:
4          print(f"Count is: {count}")
5          count = count + 1
6
7
8  count_to_n(n=4)
```

Which line of code in the code listing prevents an *infinite loop* from occurring? What would happen without it?

Common use cases of `while` loops

- **User input validation:** Prompt the user for a valid input until they give one to you!
  - *Think:* our word-guessing game example, or Wordle!
- **Game loops:** Keep a game running until some condition is met
  - Common examples: You run out of lives or attempts
- Iterating through values
  - Examples:
    - Counting from 0 to n ✅
    - Looping through every character in a string (via subscription notation)

Common use cases of `while` loops

- **User input validation:** Prompt the user for a valid input until they give one to you!
  - *Think:* our word-guessing game example, or Wordle!
- **Game loops:** Keep a game running until some condition is met
  - Common examples: You run out of lives or attempts
- Iterating through values
  - Examples:
    - Counting from 0 to n ✅
    - Looping through every character in a string (via subscription notation) ✨

```python
def reverse(a_str: str) -> str:
    """Reverse a string"""
    idx: int = 0
    result: str = ""
    while idx < len(a_str):
        result = a_str[idx] + result
        idx = idx + 1

    return result


print(reverse(a_str="abc"))
```