

Computational Thinking

Lecture 04: Conditionals & Control Flow

University of Engineering and Technology
VIETNAM NATIONAL UNIVERSITY HANOI

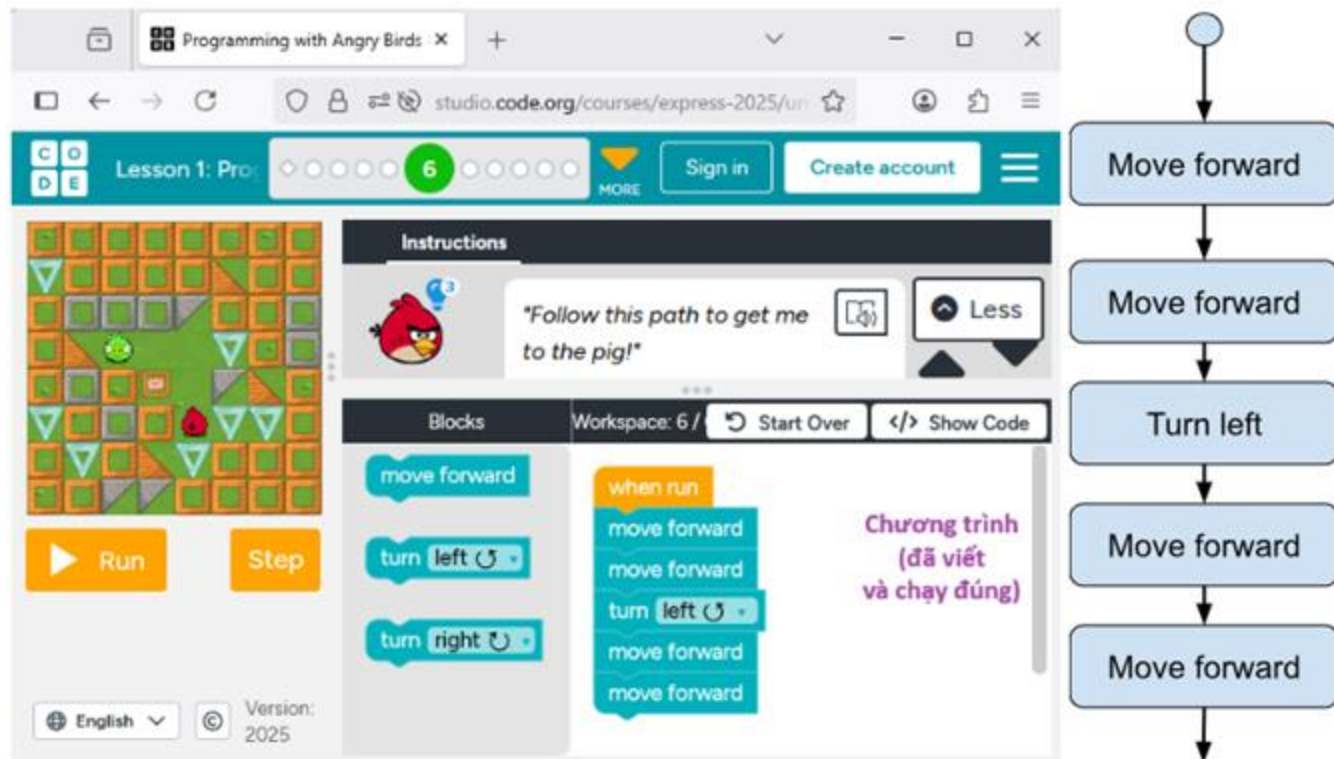
Outline

- If Statements
- Nested If Statements
- Debugging: Watches and Traces
- Testing: Code Coverage
- A Little Bit about Functions

If Statements

Program Flow

In simple programs, instructions are executed **always** in the **same order**, no matter what



But we need choices

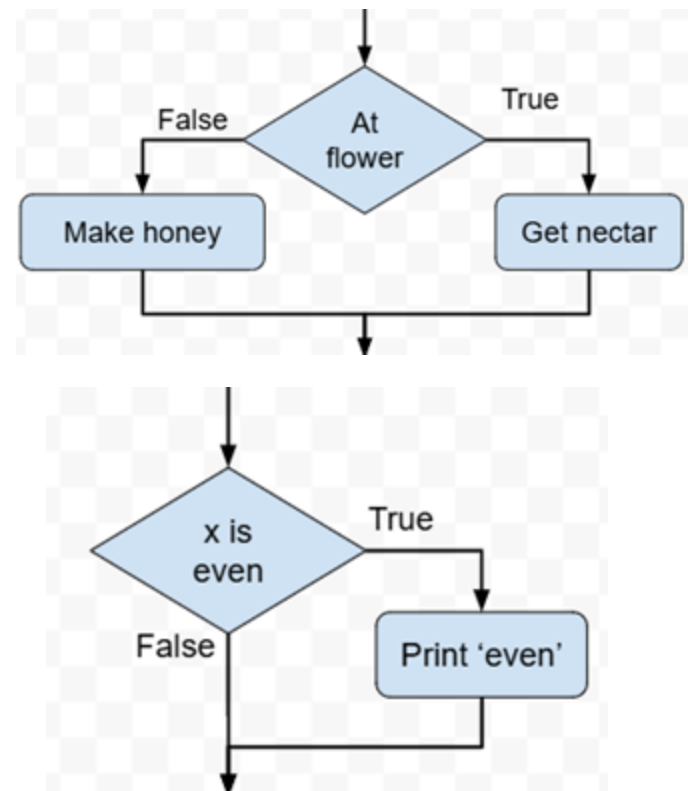
We need to make decisions!

We need a way to **control** what happens based on inputs

- If at flower, then get nectar else make honey



- If x is even, then print('even') else do nothing



If statement

We need to make decisions!

We need a way to **control** what happens based on inputs

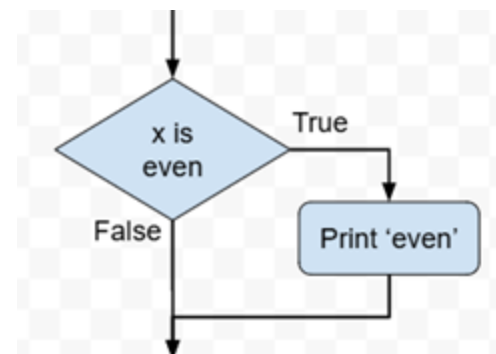
- Syntax

```
if <Boolean-expression>:  
    <statement>  
    ...  
    <statement>
```

if **<Boolean-expression>** is True,
then execute all of the statements
indented directly underneath,
until first non-indented statement

- Example

```
if x % 2 == 0:  
    print('even')
```



Examples of Boolean expressions

```
is_rainy = False
```

```
is_windy = True
```

```
temp = 12
```

Boolean variables:

```
if is_rainy:  
    print("Bring an umbrella!")
```

Boolean operations:

```
if is_windy and not is_rainy:  
    print("Let's fly a kite!")
```

Comparison operations:

```
if temp < 30 and is_rainy:  
    print("Beware of ice!")
```

```
if temp > 70:  
    print("Warm vibes!")
```

If-else statement

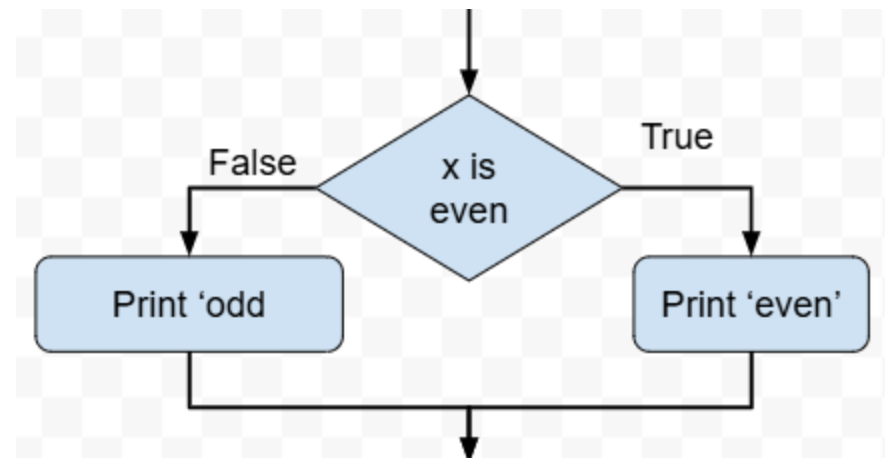
■ Syntax

```
if <Boolean-expression>:  
    <statement>  
    ...  
else:  
    <statement>  
    ...
```

■ Example

```
if x % 2 == 0:  
    print('even')  
else:  
    print('odd')
```

if **<Boolean-expression>** is True,
then execute statements indented
under **if**;
otherwise (**<Boolean-expression>** is
False) execute the statements
indented under **else**



Structure vs. Flow

Program Structure

- How code is **presented** in file
 - Which lines are earlier or later in file
 - Which lines are indented inside function or if
- Defines possibilities over **multiple executions**

Program Flow

- Aka **control flow**
- Order in which lines of code are **executed**
 - Not the same as structure
 - Some statements might be skipped
 - Some statements (in function body) might be executed many times
- Defines what happens in a **single execution**

Example

Program Structure

```
# odd_even.py
x = 10
if x % 2 == 0:
    print('even')
else:
    print('odd')
```

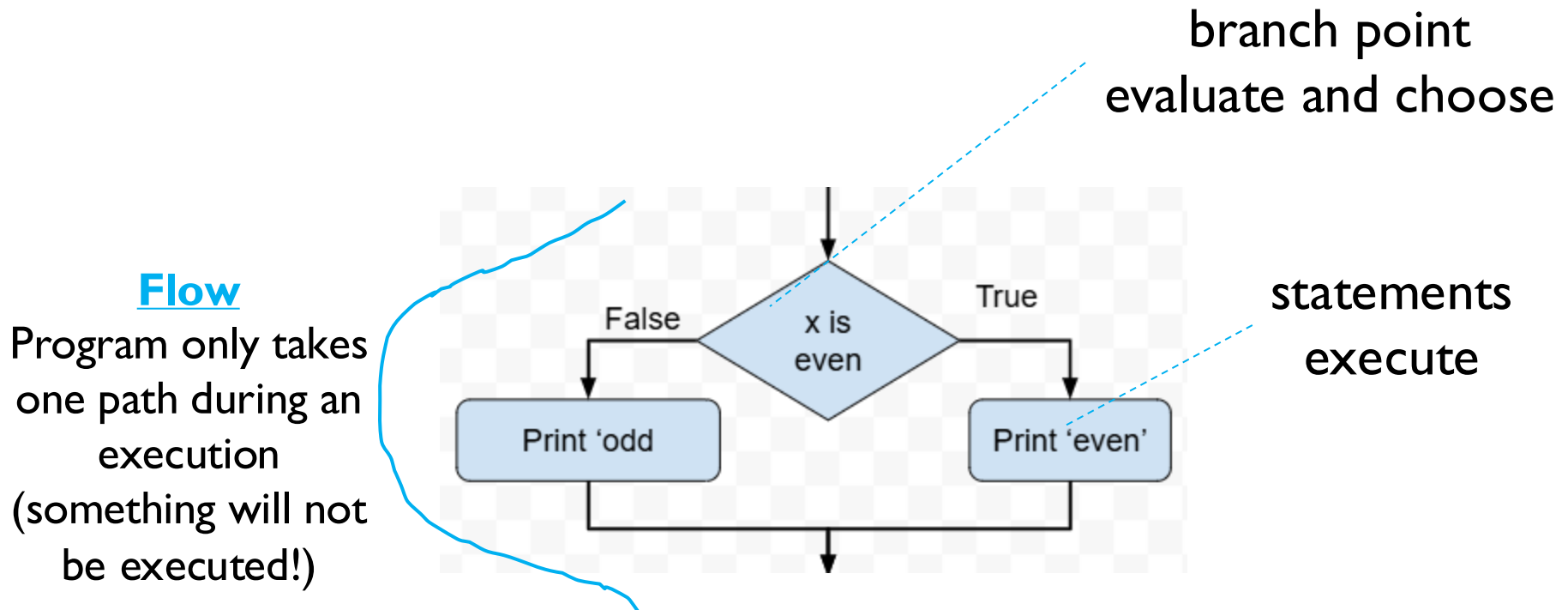
print('odd') occurs once in
the program **structure**

Program Flow

```
> python odd_even.py
even
```

program **flow** causes
print('odd') to execute
ZERO time

If Statements Affect Control Flow



Diagrams like this are called **flowcharts**

Program Flow and Variables

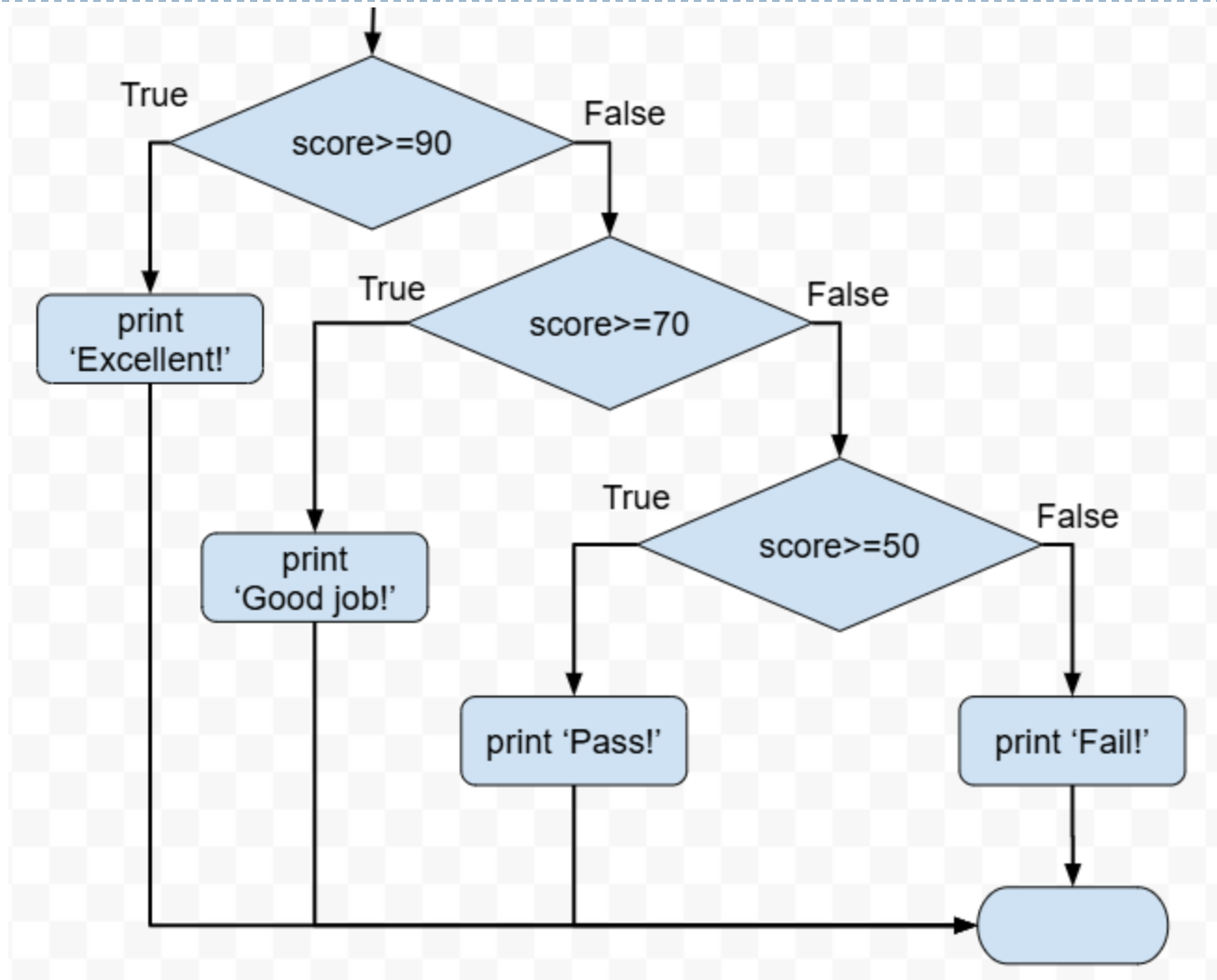
Variables created inside the body of an **if** continue to exist past the **if**:

```
a = 0
if a == 0:
    b = a + 1
print(b)
```

b is created inside this block
if the assignment is executed

But variable creation occurs only if the program actually executes the assignment that does the creation

Multiple choices



if-elif-else statements

Syntax

```
if <Boolean expression>:  
    <statement>  
    ...  
elif <Boolean expression>:  
    <statement>  
    ...  
...  
else:  
    <statement>  
    ...
```

Example

```
if score >= 90:  
    print("Excellent!"  
    )  
elif score >= 70:  
    print("Good job!")  
elif score >= 50:  
    print("Pass.")  
else:  
    print("Fail.")
```

if-elif-else statements

Syntax

```
if <Boolean expression>:  
    <statement>  
    ...  
elif <Boolean expression>:  
    <statement>  
    ...  
...  
else:  
    <statement>  
    ...
```

Notes on use

- No limit on number of **elif**
 - Must be between **if**, **else**
- **else** is optional
 - if-elif by itself is fine
- Booleans checked in order
 - Once Python finds a true **<Boolean-expression>**, skips over all the others
 - else means **all** **<Boolean-expression>** are false

If vs. If-Elif

Series of If Statements

```
x = 0
if x == 0:
    print("x is 0!")
if x == 0:
    print("still 0!")
if x == 0:
    print("even still 0")
```

```
x is 0!
still 0!
even still 0
```

vs. If-Elif Statements

```
x = 0
if x == 0:
    print("x is 0!")
elif x == 0:
    print("still 0!")
elif x == 0:
    print("even still 0")
```

```
x is 0!
```

Nothing else gets printed!

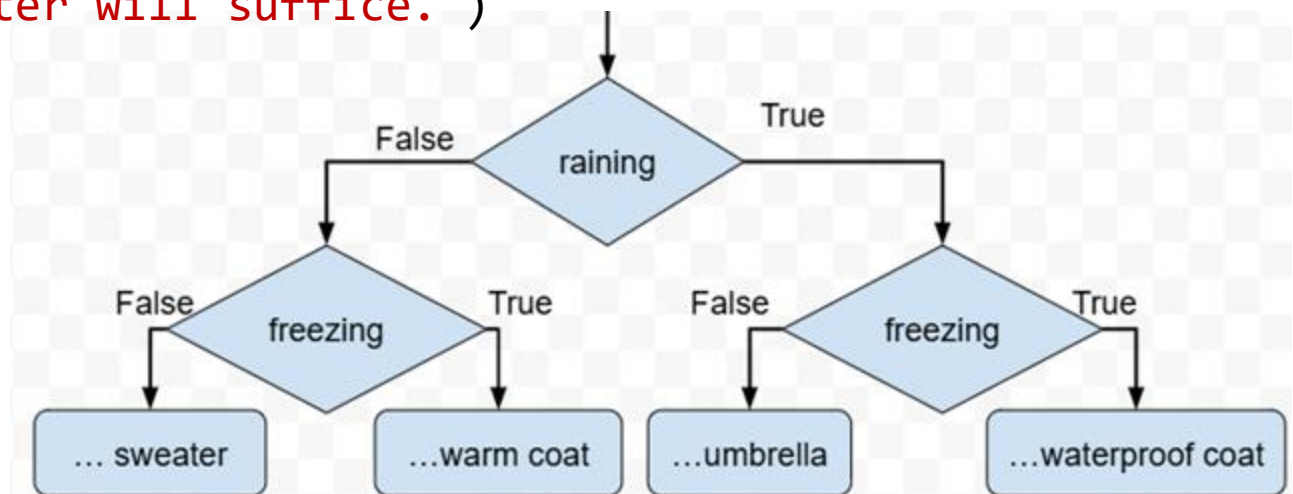
Nested if statements

The logic can get more complicated

```
# what to wear?
if raining and freezing:
    print('Wear a waterproof coat.')
elif raining and not freezing:
    print('Bring an umbrella.')
elif not raining and freezing:
    print('Wear a warm coat!')
else:
    print('A sweater will suffice.')
```

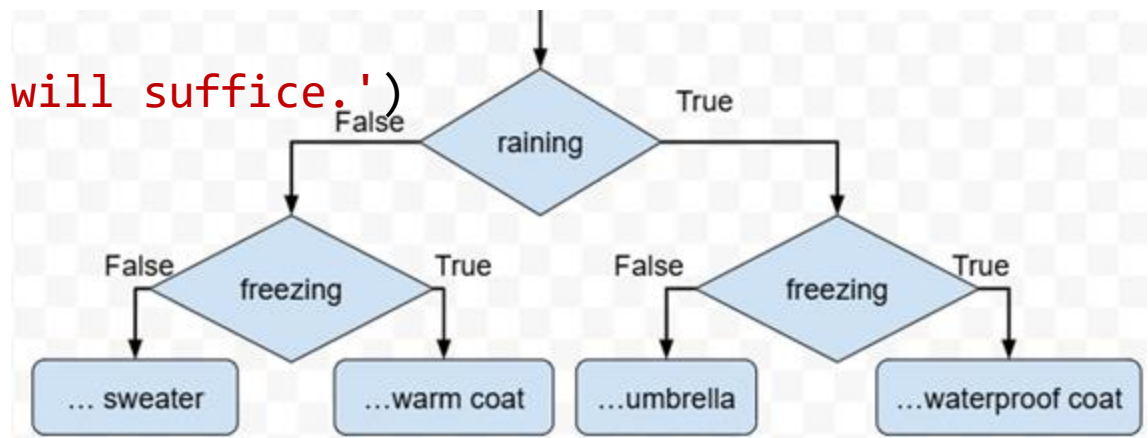
The logic can get more complicated

```
if raining and freezing:  
    print('Wear a waterproof coat.')  
elif raining and not freezing:  
    print('Bring an umbrella.')  
elif not raining and freezing:  
    print('Wear a warm coat!')  
else:  
    print('A sweater will suffice.')
```



Nested conditions

```
if raining:
    if freezing:
        print('Wear a waterproof coat.')
    else:
        print('Bring an umbrella.')
else:
    if freezing:
        print('Wear a warm coat!')
    else:
        print('A sweater will suffice.')
```



Debugging: Watches and Traces

Debugging flow with print statements

```
# determine winner
```

```
if x_score > y_score:  
    winner = "x"  
else:  
    winner = "y"
```

Can use **print** statements
to examine program flow
and variable values

"Trace" prints

Can use **print** statements to examine program flow and variable values

```
# determine winner
print('before if')
if x_score > y_score:
    print('inside if')
    winner = "x"
else:
    print('inside else')
    winner = "y"
print('after if')
```

"trace" print statements

before if
inside if
after if

x_score must have been greater than y_score

"Trace" vs. "Watch" Prints

```
# determine winner
```

```
--> print('before if')
if x_score > y_score:
--> print('inside if')
    winner = "x"
--> print('winner = '+winner)
else:
--> print('inside else')
    winner = "y"
--> print('winner = '+winner)
--> print('after if')
```

--> **Traces**

trace program flow

*What code is being executed?
Place print statements at the
beginning of a code block that
might be skipped.*

--> **WATCHES**

watch **data** values

*What is the value
of a variable?
Place print statements
after assignment statements.*

Testing: Code Coverage

Inspiration for Test Cases

- Previous sources of inspiration:
 - Rule of 1, 2, 0
 - Common and edge cases
- With if statements, a new source:
 - Invent enough test cases to cause every if-elif-else body to be executed
 - Including nested if statements

Testing of If Statement

```
if score >= 90:  
    print("Excellent!") ←  
elif score >= 70:  
    print("Good job!") ←  
elif score >= 50:  
    print("Pass.") ←  
else:  
    print("Fail.") ←
```

Invent four test cases,
one for each possible
place control flow
could reach
E.g., 91, 80, 55, 40

Testing of Nested If Statement

```
if raining:
    if freezing:
        print('Wear a waterproof coat.') ←
    else:
        print('Bring an umbrella.') ←
else:
    if freezing:
        print('Wear a warm coat!') ←
    else:
        print('A sweater will suffice.') ←
```

Again need four test cases: each possible combination of True and False

Black Box vs. Glass Box Testing

Black Box


- The function is "opaque"
- Invent test cases based solely on the specification

Glass Box

- We can "see inside" the function
- Invent test cases based also on the code that implements the function
- What we just added with examination of the code of if statements

A little bit about functions

Remember this game?






main.py	Run	Output
<pre> 1 def move_forward(): 2 print('move forward') 3 def turn_left(): 4 print('turn_left') 5 6 move_forward() 7 move_forward() 8 move_forward() 9 turn_left() 10 move_forward() 11 move_forward() 12 move_forward() </pre>		<pre> move forward move forward move forward turn_left move forward move forward move forward === Code Execution Successful </pre>

The little Scratch game's instruction set already include **move forward** and **turn left**. Python's hasn't.

So we define them ourselves. They are **functions**

A little bit about functions

function definitions

main.py		Run	Output
<pre> 1 def move_forward(): 2 print('move forward') 3 def turn_left(): 4 print('turn_left') 5 6 move_forward() 7 move_forward() 8 move_forward() 9 turn_left() 10 move_forward() 11 move_forward() 12 move_forward() </pre>	  		<pre> move forward move forward move forward turn_left move forward move forward move forward </pre> <p>=== Code Execution Success</p>

function calls

Each call to a function
executes the function once

Functions with arguments

```
def what_to_wear(raining, freezing):  
    if raining:  
        if freezing:  
            ...  
    else:  
        if freezing:  
            ...
```

```
what_to_wear(True, False)
```

```
what_to_wear(True, True)
```

```
what_to_wear(False, False)
```

```
what_to_wear(raining=False, freezing=True)
```

what_to_wear can be called with different sets of values for raining and freezing. They are arguments

When you invent these tests without knowing the code inside `what_to_wear`, it's blackbox testing

Functions that return values

```
def what_to_wear(raining, freezing):  
    if raining:  
        if freezing:  
            return 'Waterproof coat'  
        else:  
            return 'Umbrella'  
    else:  
        if freezing:  
            return 'Warm coat'  
        else:  
            return 'Sweater'
```

```
today_outfit = what_to_wear(True, False)  
sunny_day_outfit = what_to_wear(False, False)
```

this kind of functions are usually called to calculate some value before assign it to a variable for later use

Calling functions of a module

- In another module

```
from sample_functions import what_to_wear
```

```
today_outfit = what_to_wear(True, False)
```

File sample_functions.py

```
def what_to_wear(raining, freezing):  
    if raining:  
        if freezing:  
            return 'Waterproof coat'  
        else:  
            return 'Umbrella'  
    else:  
        ...
```

Calling functions of a module

- Or in interactive mode

```
>>> from sample_functions import what_to_wear
```

```
>>> print_what_to_wear(True, False)
```

Wear a waterproof coat.

```
>>>
```

File sample_functions.py

```
def print_what_to_wear(raining,
freezing):
    if raining:
        if freezing:
            print('Wear a waterproof
coat.')
        else:
            print('Bring an umbrella.')
    else:
        ...
```

A little bit about functions

Functions are like building blocks of programs
(more on that later).

But from now on, we will use them a lot

Summary - Key Takeaways

- **If** statements control the program's flow.
 - Nested **If** conditions for more **complex** logic
- Program structure **differs** from program flow.
- **Test** every branch of your code.
- Use **prints** to debug program flow.
 - Trace versus Watch
- **Functions** are like **building blocks** of programs