Philipps Universität Marburg

Introduction to Python

Monday, August 29 · 3:45 – 5:30pm

Some of the slides in this slide deck are adapted from Wells Santo (AI4ALL), Laura Biester & Jule Schatz (University of Michigan)

Remember these?

- Variables
- Conditional Statements
- Loops

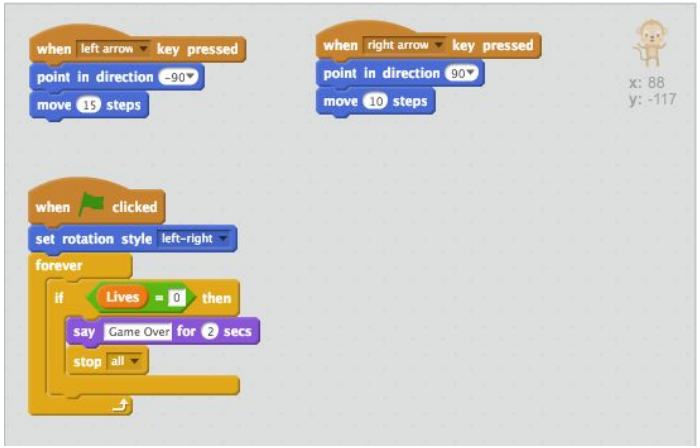
Now, you will learn how to code these concepts and more in **Python!**

What is Python?

- **Python** is a *programming language*.
- A **programming language** defines what actions we can tell a computer and exactly how to tell the computer what action to do

What is Python?

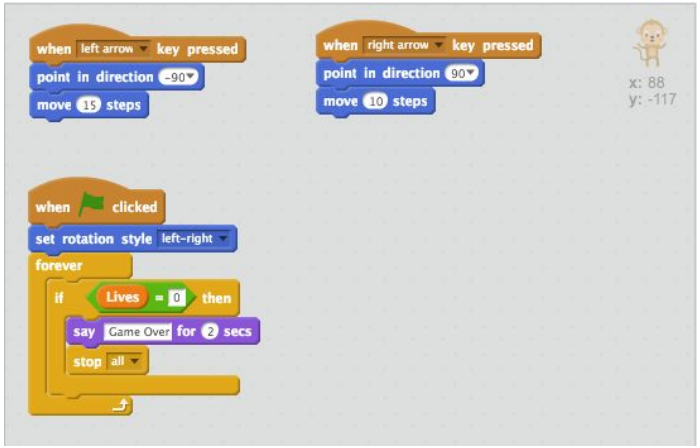
- **Python** is a *programming language*.
- A **programming language** defines what actions we can tell a computer and exactly how to tell the computer what action to do



Is this a programming language?

What is Python?

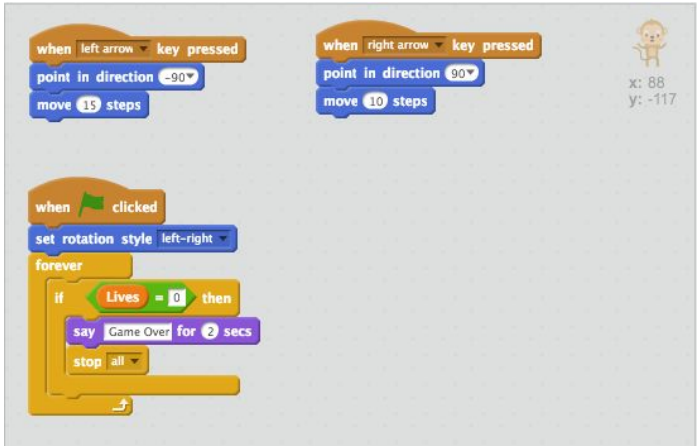
- **Python** is a *programming language*.
- A **programming language** defines what actions we can tell a computer and exactly how to tell the computer what action to do



Is this a programming language? **YES**

What is Python?

- **Python** is a *programming language*.
- A **programming language** defines what actions we can tell a computer and exactly how to tell the computer what action to do



Other popular languages today: JavaScript, C++, Golang, R, Java, Swift

Familiar with any of these?

Coding concepts we will cover

- Variables
- Simple datatypes
- Loops
- Conditional statements
- Comparison operators
- Mathematical operations
- Order of operations

Coding concepts we will cover

- ✓ Variables
- ✓ Simple datatypes
- ❑ Loops
- ❑ Conditional statements
- ❑ Comparison operators
- ❑ Mathematical operations
- ❑ Order of operations

You already know these in Python!

Coding concepts we will cover

- ✓ Variables
- ✓ Simple datatypes
- ⇝ Loops
- ⇝ Conditional statements
- ❑ Comparison operators
- ❑ Mathematical operations
- ❑ Order of operations

You already know these in Python!

You understand these concepts and have used them in Scratch!

Coding concepts we will cover

- ✓ Variables
- ✓ Simple datatypes
- ⇝ Loops
- ⇝ Conditional statements
- 💡 Comparison operators
- 💡 Mathematical operations
- 💡 Order of operations

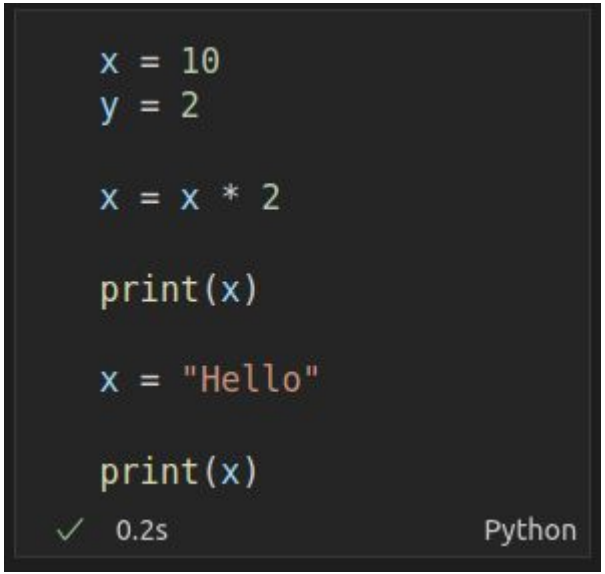
You already know these in Python!

You understand these concepts and have used them in Scratch!

You might not know this yet, but you are probably already an expert in these concepts ;)

Variables (refresh)

- Just like in math, we can use variables to represent other values.
- Variables hold values that can be overwritten and updated.



Types of Variables (refresh)

Numbers

Integers:

x = 10

y = 2

Floats:

x = 15.5

y = 3.14

Strings

Sequence of *characters* surrounded by “ “ or ‘ ‘

Characters?

x = ‘b’

y = ‘a’

Strings?

my_string = “Hello, World!”

new_string = x + y

Variables exercises

In VSCode, write the code for the exercises below. Output the value of x for each exercise by using the **print()** function.

Exercise 1:

```
# Make a variable x and set it equal to 200


print(x) # print x!
```

Exercise 2:

```
# Add 2 to x to update x to 202
```

Exercise 3:

```
# Overwrite x to represent the string "Hello, world!"
```

Naming variables: 3 things to remember

(1) Variable names should start with a letter or _

```
# Will this work?
4ever = 100000000
print(4ever)
```

```
# NO!
4ever = 100000000
print(4ever)
```

0.3s Python

```
  File "/tmp/ipykernel_48747/1299897248.py", line 1
    4ever = 100000000
    ^
SyntaxError: invalid syntax
```

```
# Will this work?
for_ever = 100000000
print(for_ever)
```

```
# YES!
for_ever = 100000000
print(for_ever)
```

✓ 0.2s

```
100000000
```

```
# Will this work?
_4ever = 100000000
print(_4ever)
```

```
# YES!
_4ever = 100000000
print(_4ever)
```

✓ 0.2s

```
100000000
```

Naming variables: 3 things to remember

(2) Variables are case-sensitive!
(Capitalization matters!)

```
# What will be the output?
cool_Variable = 2.1
cool_variable = "Hola :)"
print(cool_Variable)
```

Naming variables: 3 things to remember

(3) You can’t have spaces in variable names

```
# Will this work?
cool variable = "Hola :)"
```

```
# NO!!
cool variable = "Hola :)"
```

⊗ 0.2s Python

```
  File "/tmp/ipykernel_48747/588796287.py", line 2
    cool variable = "Hola :)"
         ^
SyntaxError: invalid syntax
```

Check-in!

What are the **datatypes** (types of variables) of var1, var2, and var3?

```
var1 = "Game Over :("
var2 = 200
var3 = 20.0
```

Check-in!

What are the **datatypes** (types of variables) of var1, var2, and var3?

```
var1 = "Game Over :(" # string
var2 = 200 # integer
var3 = 20.0 # float
```

```
# Try this:
var1 = "Game Over :("
type(var1)
```

Math

Math is so easy with Python!

+

Addition

```
# x equals 1 plus 1
x = 1 + 1
print(x)
```

-

Subtraction

```
# x equals -15.2 minus -31
x = -15.2 - 31
print(x)
```

*

Multiplication

```
# x = eight times eight
x = 8 * 8
print(x)
```

/

Division

```
# x = 3 divided by 4
x = 3 / 4
print(x)
```

Math

Math is so easy with Python!

**

Exponentiation

```
# x = 8 squared
x = 8 ** 2
print(x)
```

%

Modulo

```
# x = remainder of 10 / 3
x = 10 % 3
print(x)
```

Order of Operations

Python follows the order of operations.

```
# add 1 to 8 squared
x = 8 * 8 + 1
print(x)
```

is the same as

```
# add 1 to 8 squared
x = 8 ** 2 + 1
print(x)
```

```
# what will this be?
x = 8 ** (2 + 1)
print(x)
```

```
# no implicit multiplication with parentheses
x = 8(8)
print(x)
```

Coding concepts

- ✓ Variables
- ✓ Simple datatypes
- ❑ Loops
- ❑ Conditional statements
- ❑ Comparison operators
- ✓ Mathematical operations
- ✓ Order of operations

Almost there!

Comparison Operators

We use comparison operators to check if conditionals are **True** or **False**

What operator do we use to check if two values are equal?

Two equal signs! ==

💡 New datatype! **Boolean**

A Boolean variable can only represent two options: True or False

```
# Boolean?
what_is_this = True
type(what_is_this)
```

```
# What happens?
what_is_this = True
i_am_true = 1
what_is_this == i_am_true
```

Comparison Operators

We use comparison operators to check if conditionals are **True** or **False**

What operator do we use to check if two values are equal?

Two equal signs! ==

False equals 0 and True can be anything except 0

💡 New datatype! **Boolean**

A Boolean variable can only represent two options: True or False

```
# Boolean?
what_is_this = True
type(what_is_this)
```

```
# What happens?
what_is_this = True
i_am_true = 1
what_is_this == i_am_true
```

Comparison Operators

We use comparison operators to check if conditionals are **True** or **False**

==	Equal	!=	Not Equal
<	Less Than	<=	Less Than or Equal To
>	Greater Than	>=	Greater Than or Equal To

Comparison Operators

```
# What does this print?
x = 30
y = 15
print(x == y)
```

True or False

Comparison Operators

```
# What does this print?
x = 30
y = 15
print(x == y)
```

True or **False**

```
# What does this print?
x = 30.0
y = 15 * 2
print(x == y)
```

True or False

Comparison Operators

```
# What does this print?
x = 30
y = 15
print(x == y)
```

True or **False**

```
# What does this print?
x = 30.0
y = 15 * 2
print(x == y)
```

True or False

```
# What does this print?
x = (2 - 3) * 5
print(x > 0)
```

True or False

Comparison Operators

```
# What does this print?
x = 30
y = 15
print(x == y)
```

True or **False**

```
# What does this print?
x = 30.0
y = 15 * 2
print(x == y)
```

True or False

```
# What does this print?
x = (2 - 3) * 5
print(x > 0)
```

True or **False**

```
# What does this print?
x = (2 - 3) * 5
print(x < 0)
```

True or False

Comparison Operators

```
# What does this print?
x = 30
y = 15
print(x == y)
```

True or **False**

```
# What does this print?
x = 30.0
y = 15 * 2
print(x == y)
```

True or False

```
# What does this print?
x = (2 - 3) * 5
print(x > 0)
```

True or **False**

```
# What does this print?
x = (2 - 3) * 5
print(x < 0)
```

True or False

Comparison Operators

```
# What does this print?
x = (2 - 3) * 5
print(x != 0)
```

True or False

Comparison Operators

```
# What does this print?
x = (2 - 3) * 5
print(x != 0)
```

True or False

```
# What does this print?
i_am_true = True
i_am_false = False
print(i_am_true >= i_am_false)
```

True or False

Comparison Operators

```
# What does this print?
x = (2 - 3) * 5
print(x != 0)
```

True or False

```
# What does this print?
i_am_true = True
i_am_false = False
print(i_am_true >= i_am_false)
```

True or False

```
# What does this print?
big_number = 1000
small_number = 2
print(small_number <= big_number)
```

True or False

Comparison Operators

```
# What does this print?
x = (2 - 3) * 5
print(x != 0)
```

True or False

```
# What does this print?
i_am_true = True
i_am_false = False
print(i_am_true >= i_am_false)
```

True or False

```
# What does this print?
big_number = 1000
small_number = 2
print(small_number <= big_number)
```

True or False

```
# What does this print?
big_number = 1000
small_number = 2
print(small_number * 500 + 1 <= big_number)
```

True or False

Comparison Operators

```
# What does this print?
x = (2 - 3) * 5
print(x != 0)
```

True or False

```
# What does this print?
i_am_true = True
i_am_false = False
print(i_am_true >= i_am_false)
```

True or False

```
# What does this print?
big_number = 1000
small_number = 2
print(small_number <= big_number)
```

True or False

```
# What does this print?
big_number = 1000
small_number = 2
print(small_number * 500 + 1 <= big_number)
```

True or **False**

Conditional statements (review)

Code that executes only when a certain condition is met

In English:

If there are apples in the basket,

announce the number of apples.

3 apples!

Conditional Statements (comparison exercise)

Exercise:

```
apples_in_basket = 3

"""
How can we check if the basket has apples?

Use a comparison operator to check
if the basket has apples
"""
```

Conditional Statements (solutions)

Exercise:

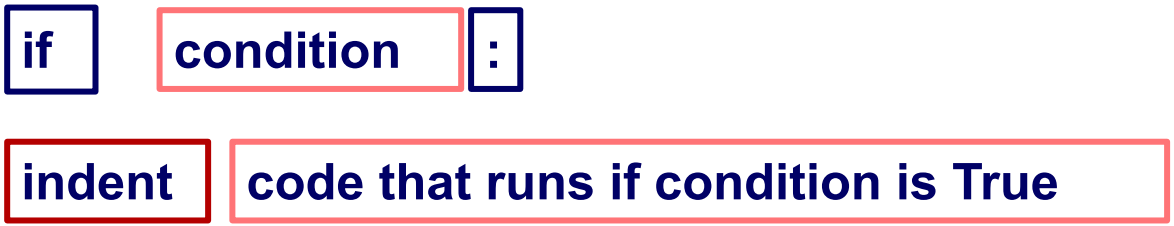
```
apples_in_basket = 3

print(apples_in_basket != 0)

# or


print(apples_in_basket > 0)
```

Conditional statements (if)



Indent: Can be a tab or 4 spaces. Just be consistent!

Conditional statements (if)

```
if apples_in_basket > 0 :
indent print(apples_in_basket)
```

Exercise:

```
# Try it out!
apples_in_basket = 3

# Write the Python code:
```

Conditional statements (if..else)

What if we want to do something otherwise?

```
if condition :
    indent code that runs if condition is True
else :
    indent code that runs if condition is False
```

Conditional statements (if..else)

```
if apples_in_basket > 0 :
indent print(apples_in_basket)
else :
indent print("No more apples :(")
```

```
# Try it out!
apples_in_basket = 3

# Write the Python code for the "else"
if apples_in_basket > 0:
    print(apples_in_basket)
```

Conditional Statements (elif)

What if we want to run code for more than these two conditions?

Condition 1: If there are more than one apple in the basket, print “there **are** x apple**s** in the basket”

Condition 2: If there is one apple in the basket, print “there **is one** apple in the basket”

Condition 3: If the basket is empty, print “there are no apples in the basket”

Conditional Statements (elif)

What if we want to run code for more than these two conditions?

Condition 1: If there are more than one apple in the basket, print “there **are** x apple**s** in the basket”

if

Condition 2: If there is one apple in the basket, print “there **is one** apple in the basket”

elif (stands for “else if”)

Condition 3: If the basket is empty, print “there are no apples in the basket”

else

Conditional statements (elif)

```
if    Condition 1    :
      code that runs if
      condition 1 is True
elif  Condition 2    :
      code that runs if
      condition 1 is True
else  :
      code that runs if
      condition 3 is True
```

Exercise: Write Python code based on the template to the left that handles the 3 conditions of the apple basket.

1. When the basket has more than one apple, print “there are x apples in the basket”
2. When the basket has only one apply, print “there is one apple in the basket”
3. If the basket is empty, print “there are no apples in the basket”

Test each condition by changing the value of `apples_in_basket`

Conditional statements (elif)

Possible solution:

```
apples_in_basket = 0

if apples_in_basket > 1:
    print("there are", apples_in_basket, "apples in the basket")
elif apples_in_basket == 1:
    print("there is one apple in the basket")
else:
    print("No more apples :(")
```

Compound Conditionals

- We can also use the operators **and** and **or** in our conditions

- What would the following print?

```
num = 50
if num > 0 and num < 100:
        print("This is a number between 0 and 100!")

favorite_subject = "CS"
if favorite_subject == "CS" or favorite_subject == "math":
        print("You like STEM!")
```

Compound Conditionals

We can also use the operators **and** and **or** in our conditions

```
num = 50
if num > 0 and num < 100:
    print("This is a number between 0 and 100!")
```

Loops (review)

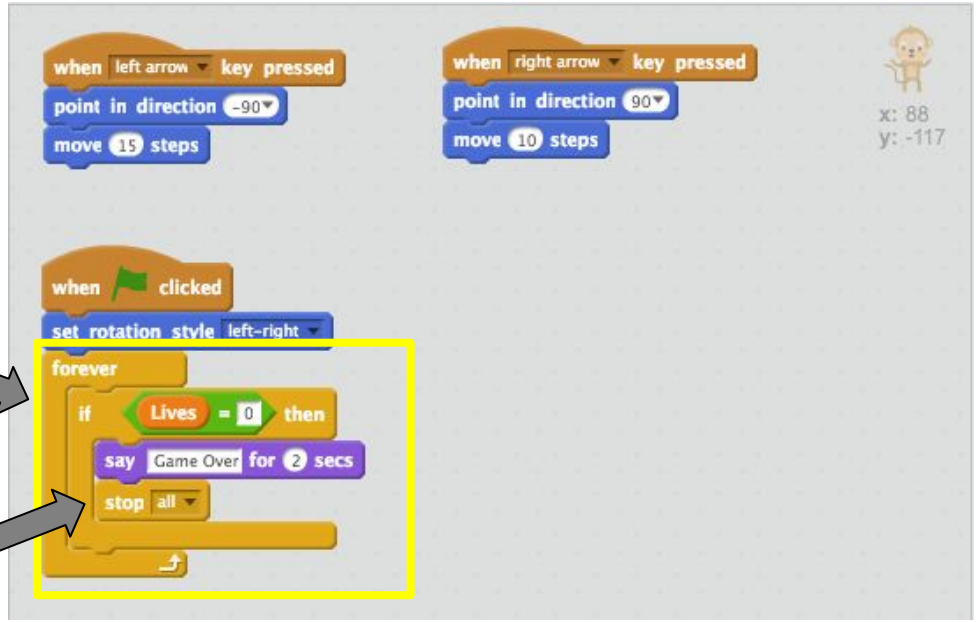
- Loops are used in coding for tasks that require repetition
- Loops use conditional statements!

While the basket of apples is not empty, take out an apple, and count it.

Loops (review)

The condition for continuing the loop is based on the value of lives

Lives = 0 is the condition for ending the loop

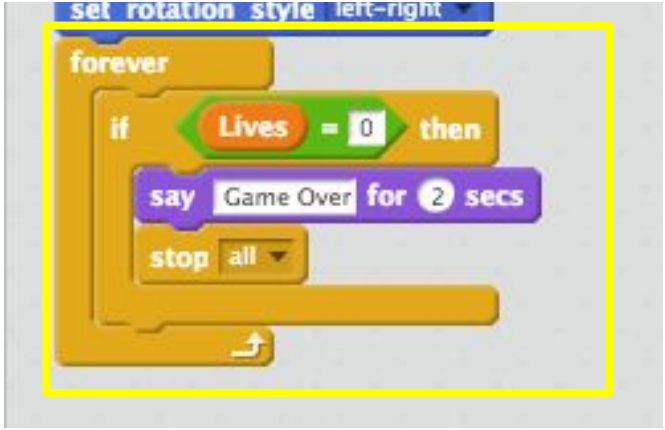


Loops

We could try to translate Scratch to Python as directly as we can:

```
Lives = 3
forever = True # make a Boolean variable "forever" and set to True

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stops the loop
```



Loops: Infinite Loops

There is a problem with this code

```
Lives = 3
forever = True # make a Boolean variable
"forever" and set to True

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stops the loop
```

Infinite Loops: This loop will run *infinitely* because the value of Lives is never updated within this **scope,** Lives will never equal 0, and the **break** will never be called to stop the loop.

Loops: Infinite Loops

There is a problem with this code

```
Lives = 3
forever = True # make a Boolean variable
"forever" and set to True

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stops the loop
```

Infinite Loops: This loop will run *infinitely* because the value of Lives is never updated within this **scope,** Lives will never equal 0, and the **break** will never be called to stop the loop.

When writing loops, we must make sure the condition to end the loop is **reachable.** To stop the loop, **forever** must be changed to false, or Lives must be changed to 0.

Loops: Infinite Loops

There is a problem with this code

```
Lives = 3
forever = True # make a Boolean variable
"forever" and set to True


while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stops the loop
```

Infinite Loops: This loop will run *infinitely* because the value of Lives is never updated within this **scope,** Lives will never equal 0, and the **break** will never be called to stop the loop.

When writing loops, we must make sure the condition to end the loop is **reachable.** To stop the loop, **forever** must be changed to false, or Lives must be changed to 0.

Note: The code works in scratchy because there is other code that updates the value of Lives that is running at the same time. You will learn more about game loops later this week, but the main takeaway now is, the stopping condition must be reached!

Loops: Fixing the infinite loop

```
Lives = 3
forever = True

while forever:
    if Lives == 0:
        print("Game Over")
        break # stop
    # update the value of Lives
    Lives = Lives - 1
```

Here, the stopping condition will be reached since the value of Lives will reach 0

Loops: Iterations

```
Lives = 3
forever = True
num_iterations = 0

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stop
    # update the value of Lives
    Lives = Lives - 1
    num_iterations = num_iterations + 1
print("The loop ran", num_iterations, "times")
```

An **iteration** is one pass through the loop.

What will this code print?

Loops: Iterations

```
Lives = 3
forever = True
num_iterations = 0

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stop
    # update the value of Lives
    Lives = Lives - 1
    num_iterations = num_iterations + 1
print("The loop ran", num_iterations, "times")
```

An **iteration** is one pass through the loop.

What will this code print?

```
Game Over
The loop ran 3 times
```

The loop ran for 3 iterations

Loops and Better Python

DE: Ich denke ich spinne!
EN: I think I spider!

Just like in natural language, direct translations from one programming language to another do not make sense or are not phrased as nicely :)

Loops and Better Python

```
Lives = 3
# We do not need "forever"
# forever = True
num_iterations = 0

while True: # We can just say "while True"
  if Lives == 0:
      print("Game Over")
      break
  Lives = Lives - 1
  num_iterations = num_iterations + 1
print("The loop ran", num_iterations, "times")
```

Loops and Better Python

```
Lives = 3
num_iterations = 0

# Instead of "True," we can use a conditional
while Lives != 0:
# Remove conditional inside loop
#    if Lives == 0:
#        print("Game Over")
#        break
  Lives = Lives - 1
  num_iterations = num_iterations + 1
print("The loop ran", num_iterations, "times")
```

Loops (exercise)

Write a **while loop** in Python that “removes an an apple from the basket” (decreases `apples_in_basket` by 1) in each iteration.

Make your program print the conditions we coded earlier (printing the number of apples, and printing when there are no apples left).

Remember, make sure the **stopping condition** is reachable to avoid an **infinite loop**!

Loops (exercise)

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket")
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1
```

What will this code print?

Loops (exercise)

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(" )
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
```

What will this code print?

Loops (exercise)

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(" )
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
```

What will this code print?

How come `"No more apples :("` never prints?

Loops (exercise)

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket")
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
```

What will this code print?

How come `"No more apples :("` never prints?

What are **two ways** we can modify this code so that it prints?

Loops (exercise)

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket")
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
```

What will this code print?

How come `"No more apples :("` never prints?

What are **two ways** we can modify this code so that it prints?

Loops (exercise)

```
apples_in_basket = 3

while apples_in_basket >= 0: # change the stopping condition
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(" )
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
No more apples :(
```

What will this code print?

How come `"No more apples :("` never prints?

What are **two ways** we can modify this code so that it prints?

1. Change the stopping condition

Loops (exercise)

```
apples_in_basket = 3

while apples_in_basket > -1: # change the stopping condition
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(" )
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
No more apples :(
```

What will this code print?

How come `"No more apples :("` never prints?

What are **two ways** we can modify this code so that it prints?

1. Change the stopping condition (at least 2 ways)

Loops (exercise)

```
apples_in_basket = 3

while apples_in_basket != 0: # original stopping condition
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    # remove else from loop
    apples_in_basket = apples_in_basket - 1
print("No more apples :(") # print statement outside of loop
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
No more apples :(
```

What will this code print?

How come `"No more apples :("` never prints?

What are **two ways** we can modify this code so that it prints?

1. Change the stopping condition (at least 2 ways)
2. Move the print statement **outside of the <u>scope</u> of the loop**

Coding concepts

- ✓ Variables
- ✓ Simple datatypes
- ✓ Loops
- ✓ Conditional statements
- ✓ Comparison operators
- ✓ Mathematical operations
- ✓ Order of operations

All of the concepts you learned how to code in Python today!

Coding concepts

Up next!

- ✓ Variables
- ✓ Simple datatypes - New datatype: **Lists**
- ✓ Loops - New type of loop: **For-Loops**
- ✓ Conditional statements - **Compound conditionals**
- ✓ Comparison operators
- ✓ Mathematical operations
- ✓ Order of operations

For-loops

For-loops are another type of loop you can use when you want to iterate through a specific range of values.

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

Try the code on the right in your VS Code notebook.

- What are the first and last values?
- How many values does it print?

For-loops

For-loops are another type of loop you can use when you want to iterate through a specific range of values.

Try the code on the right in your VS Code notebook.

- What are the first and last values? **0 and 9**
- How many values does it print? **10**

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

The range excludes the maximum number

Test your understanding

Complete the for-loop so that it prints only the even numbers in the range.

```
for i in range(20):
```

Hint: the **Modulo** operator returns the remainder after dividing

For-loops

You can specify the starting point of the range

```
for i in range(1, 11):
    print(i)
```

You can specify the amount to increment

```
for i in range(1, 11, 2):
    print(i)
```

You can go from large to small values

```
for i in range(11, 1, -2):
    print(i)
```

Lists

Lists can be used to keep track of many related items in one place.

A Python list can be created like a variable. You need your items to be separated by commas and inside brackets

Grocery list:

- Onions
- Tomatoes
- Rice
- Soup

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
```

Iterate over a list

We can use iterate through the list with a loop

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
for grocery_item in grocery_list:
    print(grocery_item)
```

Accessing list items

Each item in the list has an index number

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
print(grocery_list[0])
print(grocery_list[1])
print(grocery_list[2])
print(grocery_list[3])
```

Accessing list items

You can access items from the back

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
print(grocery_list[-1])
print(grocery_list[-2])
```

Adding to the list

To add something to the list

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]


grocery_list.append("Ice cream")
print(grocery_list)
# The item is "appended" to the end of the list
print(grocery_list[-1])
```

Changing items in the list

Changing values in the list

```
grocery_list = ["Onions", "Tomatoes", "Rice"]
grocery_list[0] = "Ice cream"
print(grocery_list)
```

Lists

Lists can contain different datatypes

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
grocery_list.append(5)
grocery_list.append(5.5)
print(grocery_list)
```

Iterating with for-loops

We can use iterate through the list with a loop

```
grocery_list = ["Onions", "Tomatoes", "Rice"]
for grocery_item in grocery_list:
    print(grocery_item)
```

We can iterate through any “iterable.”

```
for character in "Ice cream":
    print(character)
```

Strings are iterable!

```
for x in 497:
    print(x)


TypeError: 'int' object is not iterable
```

len()

The function **len()** can tell us the length of the iterable object (number of items in a list, number of characters in a string)

```
grocery_list = ["Onions", "Tomatoes", "Rice"]
print(len(grocery_list))
print(len("Ice cream"))
```

min() and max()

The functions **min()** and **max()** can tell us the the smallest and largest values in a list. Then we can use list.index(a_number) to get the index of the value in the list.

```
some_numbers = [5, 2, 1, 5, 7, 10, 32, 3]
smallest_number = min(some_numbers) # 1
index_of_smallest_number = some_numbers.index(smallest_number) #
2


print(some_numbers[index_of_smallest_number]) # 1
```

Nested loops

You can have loops within loops

```
grocery_list = ["Onions", "Tomatoes", "Rice"]
for i in grocery_list:
    for j in grocery_list:
        if i != j:
            print(i, j)
```

Dictionaries

- With lists, you access elements using their index number
- For many things, this might not be very useful, especially when you don’t know the index number of your element
- A dictionary allows us to access elements based on **keys**

```
my_dict = {"key": "value"}


print(my_dict["key"])
```

Dictionary syntax

1. Surround contents with curly brackets **{}**
2. The **key** is on the left of a **colon :** and the **value** is on the colon

```
my_dict = {"key": "value"}
```

Dictionary syntax

1. Surround contents with curly brackets **{}**
2. The **key** is on the left of a **colon :** and the **value** is on the colon
3. Just like in **lists**, the items in the dictionary are separated by commas

```
my_dict = {"key": "value", 'dictionary': 'Wörterbuch'}
```

Dictionaries

You can think of dictionaries similarly to word dictionaries.

```
my_dict = {"word": "a definition of that word"}
print(my_dict["word"])

en2de = {'apple': 'Apfel', 'onion': 'Zwiebel', 'dictionary':
'Wörterbuch'}
print(en2de["apple"])
print(en2de["onion"])
print(en2de["dictionary"])
```

Dictionaries

Keys and Values can be different datatypes

```
my_dict = {"xy-coordinates": [(2,3), (3,4), (4,5)], 'closest':(3,4)}
print(type(my_dict["xy-coordinates"])) # <class 'list'>
print(type(my_dict["closest"])) # <class 'tuple'>
```

```
my_dict = {3: 'three', 'three':3}
```

Dictionaries

Dictionaries are iterable. Simply iterating over the dictionary will iterate through the strings.

```
groceries = {"fruit": ["apple", "orange"],
             "vegetable": ["onion", "potato"]}


for category in groceries:
    for item in groceries[category]:
        print(item, "is a", category)
```

Dictionaries

- Iterable of keys and values with **dict.items()**
- Iterable of keys: **dict.keys()**
- Iterable of values: **dict.values()**

```
for category, item_list in groceries.items():
    print(item_list, "are", category)


print(groceries.keys())
print(groceries.values())
```

Exercise

Complete the for-loop to make the dictionary map the item of the list to the item's index in the list (keys are items and values are indices).

The enumerate function can tell you the iteration (index) you are currently on. That value is stored in the index variable, and the list item is stored in the item variable.

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
item_to_index = {}

for index, item in enumerate(grocery_list):
    """ complete the for-loop """
```

Functions

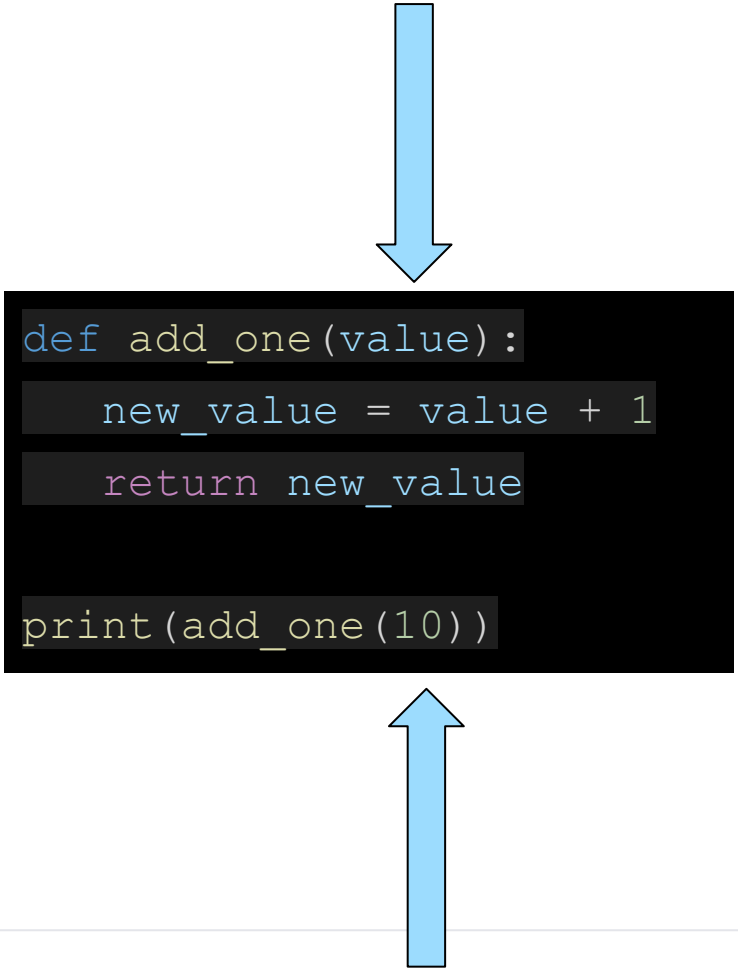
Functions are a way to reuse code

```
""" here we define the function """
def add_one(value):
    new_value = value + 1
    return new_value

""" here we use the function """
print(add_one(10))
print(add_one(2))
```

Functions: Parameters

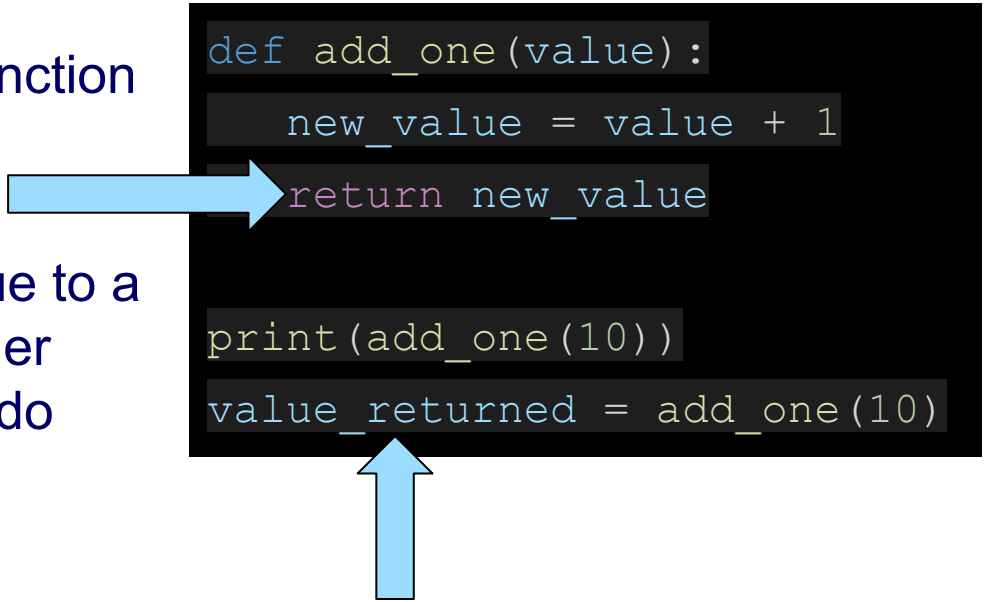
Parameters are values that you pass to a function



Functions: Return Values

Return values are values the function gives you back

You can assign the returned value to a new variable or pass them to other functions as parameters like we do with print



Functions: Return Values

Return values are values the function gives you back

Return values are not always needed

```
def my_print_function(message):
    print("Message:", message)

my_print_function("Hello!")
```

Exercise

Write (define) a function called **make_list_index**.

Parameters: The function should take in one parameter which will be a list

Function body: It should take the parameter (a list) and create a dictionary that maps the list items to their indices, just like you wrote before (below)

Return value: The function should return the dictionary

```
grocery_list = ["Onions", "Tomatoes", "Rice"]
item_to_index = {}

for index, item in enumerate(grocery_list):
    item_to_index[item] = index
```

Exercise solution

```
def make_list_index(list_in):
    item_to_index = {}
    for index, item in enumerate(list_in):
        item_to_index[item] = index
    return item_to_index

grocery_index = make_list_index(["apples", "bananas", "oranges"])
print(grocery_index)

materials_index = make_list_index(["clay", "wood", "stone"])
print(materials_index)
```

Klassen

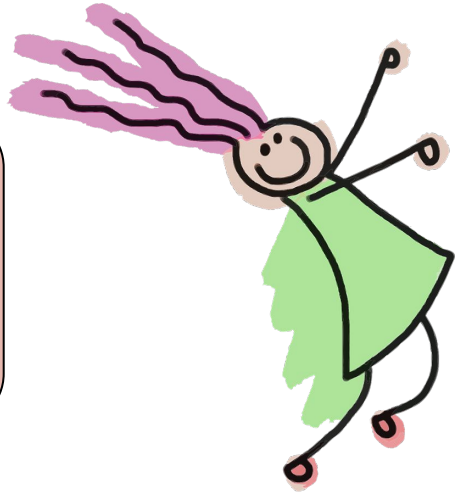
Klassen

Spielerin 1:

Name: Anna
Geburtsjahr: 2004
Haarfarbe: orange

Spielerin 2:

Name: Julia
Geburtsjahr: 2005
Haarfarbe: lila

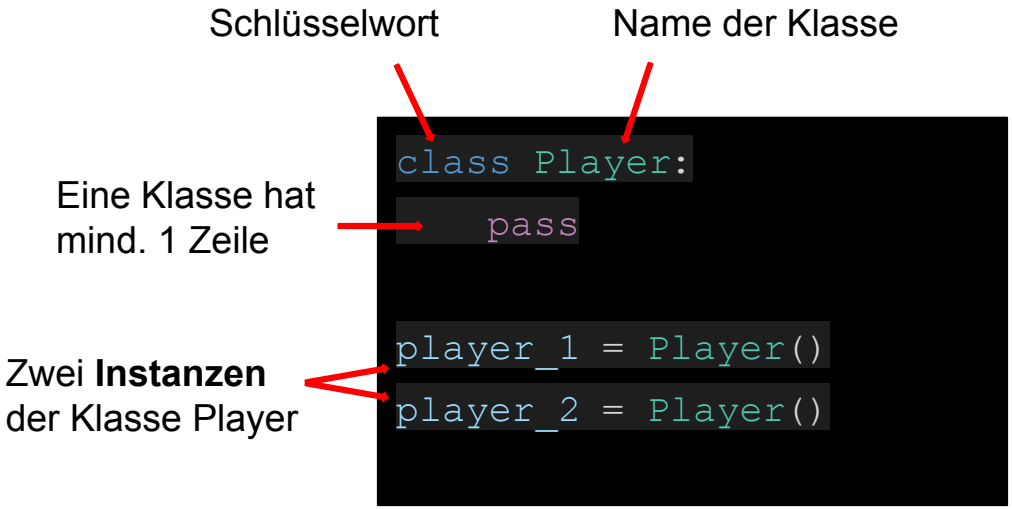


Wie können wir die beiden Spielerinnen in Python modellieren?

Klassen

- Python: *objektorientierte Programmiersprache*
  - Nahezu alles besteht aus Objekten mit gewissen Eigenschaften und Funktionn/ Funktionen
- Eine Klasse ist wie ein Bauplan für Objekte

★ Schreibe eine Klasse Player und erzeuge zwei verschiedene Player!



Aber wo sind jetzt die Eigenschaften von Anna und Julia?

Klassen

- In der `__init__` Funktion einer Klasse können Eigenschaften der Klasse implementiert werden (*Konstruktor*).
- Jede Klasse besitzt eine `__init__` Funktion, die immer ausgeführt wird, wenn eine Instanz der Klasse erzeugt wird.
- In der Funktion werden den Eigenschaften der Objekte Werte zugewiesen.
- Zugriff auf die Variablen möglich

Variablen der Klasse

Zuweisungen der Werte

```
class Player:
    def __init__(self, name, birth_year, hair_color):
        self.name       = name
        self.birth_year = birth_year
        self.hair_color = hair_color

player_1 = Player('Anna', 2004, 'orange')
player_2 = Player('Julia', 2005, 'lila')
print(player_1.name)
print(player_2.hair_color)
```

Der self-Parameter ist ein Verweis auf die aktuelle Instanz der Klasse und wird für den Zugriff auf Variablen verwendet, die zur Klasse gehören.

Klassen

★ Wir möchten von den Spielerinnen wissen, wie alt sie sind und welche Haarfarbe sie haben.

- Objekte können auch Funktionen enthalten. Funktionen in Objekten sind Funktionen, die zum Objekt gehören.

```
class Player:
    def __init__(self, name, birth_year, hair_color):
        self.name          = name
        self.birth_year    = birth_year
        self.hair_color    = hair_color

    def get_hair_color(self):
        print('Meine Haarfarbe ist ' + self.hair_color)

    def get_age(self):
        age = 2022 - self.birth_year
        print('Ich bin ' + str(age) + ' Jahre alt.')

player_1 = Player('Anna', 2004, 'orange')
player_2 = Player('Julia', 2005, 'lila')
player_1.get_age()
player_1.get_hair_color()
```

Die Funktionen greifen auf Werte einer Instanz zu (Haarfarbe, Geburtsjahr). Wir übergeben hier wieder den self-parameter!

Vererbung

- Erstelle eine Klasse Schachspieler!
- Parameter:
  - Namen
  - Geburtsjahr
  - Haarfarbe
  - Spielfarbe
- Funktionen:
  - Welche Haarfarbe
  - Welches Alter
  - Welche Spielfarbe

Vererbung

```
class Chess_player():
    def __init__(self, name, birth_year, hair_color, game_color):
        self.name           = name
        self.birth_year     = birth_year
        self.hair_color     = hair_color
        self.game_color     = game_color

    def get_hair_color(self):
        print('Meine Haarfarbe ist ' + self.hair_color)

    def get_age(self):
        age = 2022 - self.birth_year
        print('Ich bin ' + str(age) + ' Jahre alt.')

    def get_game_color(self):
        print("My game color is: " + self.game_color)
```

- Ganz schön viel Schreibarbeit!
- Die meisten Variablen und Funktionen sind die gleichen wie in der Klasse Player
- Lässt sich das irgendwie vereinfachen?

Vererbung

Erbe von der Klasse Player!

- Die Klasse Chess_Player verfügt dann ebenfalls über alle Parameter und Funktionen, wie die Klasse Player
  - Player: Eltern-Klasse (*super class*)
  - Chess_Player: Kind-Klasse (*sub class*)
- Das Schlüsselwort super() steht für „superclass“, sprich Oberklasse. Wir stellen damit eine Verbindung zwischen der Eltern-Klasse (Klasse, von der Kind erbt) und Kind-Klasse her.

Vererbung

Übersichtlicher und kürzer!

Erbe von der Klasse Player

```
class Chess_player(Player):

    def __init__(self,  name, birth_year, hair_color, game_color):
        super().__init__(name, birth_year, hair_color)
        self.game_color = game_color

    def get_game_color(self):
        print("Meine Spielfarbe ist: " + self.game_color)


chess_player_1 = Chess_player("Kim", 2003, "rot", "weiss")
chess_player_1.get_age()
chess_player_1.get_game_color()
```

Aufruf Super-Konstruktor (Konstruktor der Eltern-Klasse)

Verwendung von Methoden der Eltern-Klasse möglich