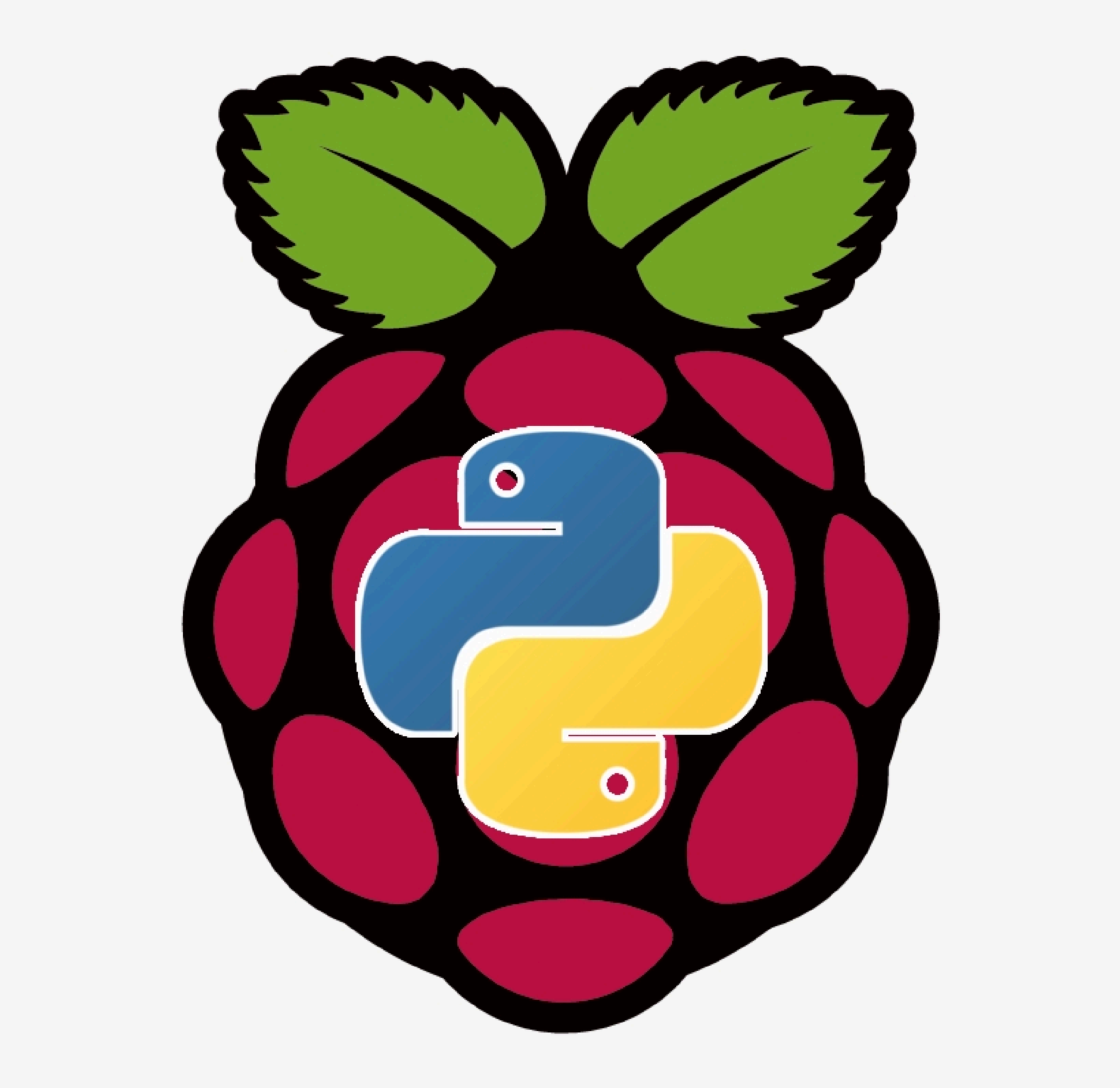
Introduction to Python 🐍

- Variables and data types
- Basic math operations
- Control flow
- Functions
- And more...

By: Hedron Hackerspace



Common Misconceptions Pt. 1

- "You need to be **smart**"
    - No you don't, just look at the instructor
- "You need to know **a lot of difficult maths**"
    - Nope, Python and libraries will do most of it for you
- "Programming is **boring**"
    - Only if you let it, there are plenty of ways to make it interesting
- "Python is **slow** compared to other languages"
    - Yes, but it depends on your implementation (ie. Numpy)

Common Misconceptions Pt. 2

- "Python is for **children**"
  - A lot of data science and cybersecurity fields use nothing *but* Python
- "Programming **isn't creative**"
  - If anything, you have to* be creative to program
- "You have to know **a lot of computer science**"
  - Really only the basics are needed

* - At least somewhat

IDEs and editors

- **IDE**: *integrated development environment*
  - It's the thing you code in
  - IDEs usually have
    - Syntax highlighting
    - Integrated terminal
    - Project tree and file tabs
- **Text Editor**: *just a text editor, lol*
  - Just meant to edit text in files, but certainly usable for coding
  - Most of the time, doesn't have any quality of life features

Desktop GUI: VS Code

Pi OS already has Python installed, so all we need is an **IDE**. We're going to use **VS Code**, but use whichever IDE you want.

- Open a terminal window and enter `sudo apt install code-oss`
- Type in your password and accept the "Are you sure?" prompt
- Once installation is complete, enter `code` in the terminal and VS Code should automatically open
  - Or you can go into the applications menu to open it
- Go into extensions and install the official Python extension
- Sign into GitHub if you have an account (not required)

Headless: Nano

Pi OS already has Python and Nano installed, so all we need to do is open it. We're going to use **Nano**, but use whichever editor you want.

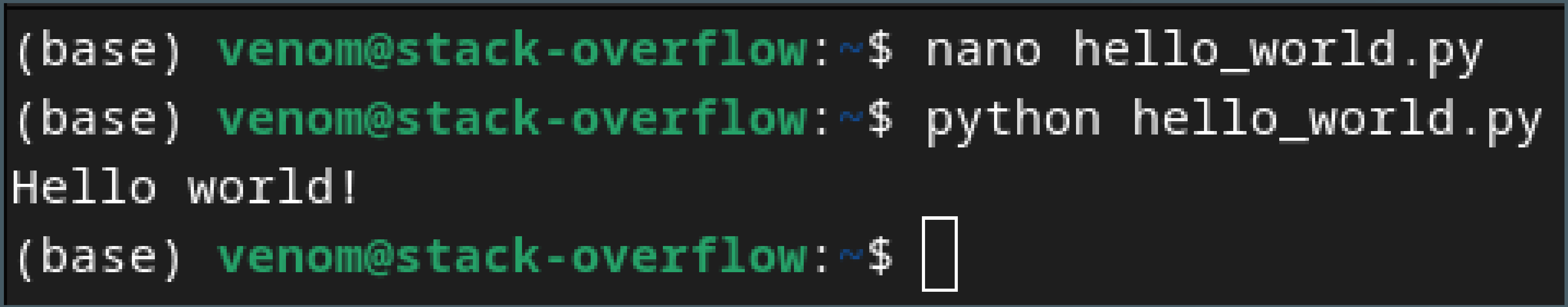
- Open a terminal window and enter `nano`
- That's it...

Creating a Python File

- Open the folder you want to keep your programs in
  - VS Code: File>Open Folder
  - nano: `cd <folder>`
- Create a Python file with the name `hello_world.py`
  - VS Code: New File>"hello_world.py"
  - nano: `nano hello_world.py` or use `touch` first
  - Python files NEED the `.py` extension for the interpreter to run
- Write the line `print("Hello world!")` and save the file
  - VS Code and nano: Ctrl+S

Running a Python File

- VS Code: Right click>Run Python>Run Python File in Terminal
- nano: Ctrl+X to exit (after you save)> `python hello_world.py`



```
(base) venom@stack-overflow:~$ nano hello_world.py
(base) venom@stack-overflow:~$ python hello_world.py
Hello world!
(base) venom@stack-overflow:~$ 
```

Congratulations, you just created and ran your first (allegedly) Python program!

3 Key Programming Concepts

(if you don't take anything else from this)

- Cooking == Programming
  - Conceptually the same thing, just one is significantly tastier
  - If you can cook or follow instructions, you can program
- EVERYTHING IS BINARY
  - This is a slightly more intermediate and won't be useful until a bit later
- The further you break something down, the easier it will be to implement

Syntax

- "Sentence structure rules" for Python
- Rules such as:
  - Python uses whitespace and indents for scope
  - Don't start names with a number, special character, etc.
  - Don't use reserved names for variables, functions, etc.
- Would not recommend learning all the rules at once
  - Better learned slowly over time
  - We will touch on things as they become relevant

Formatting

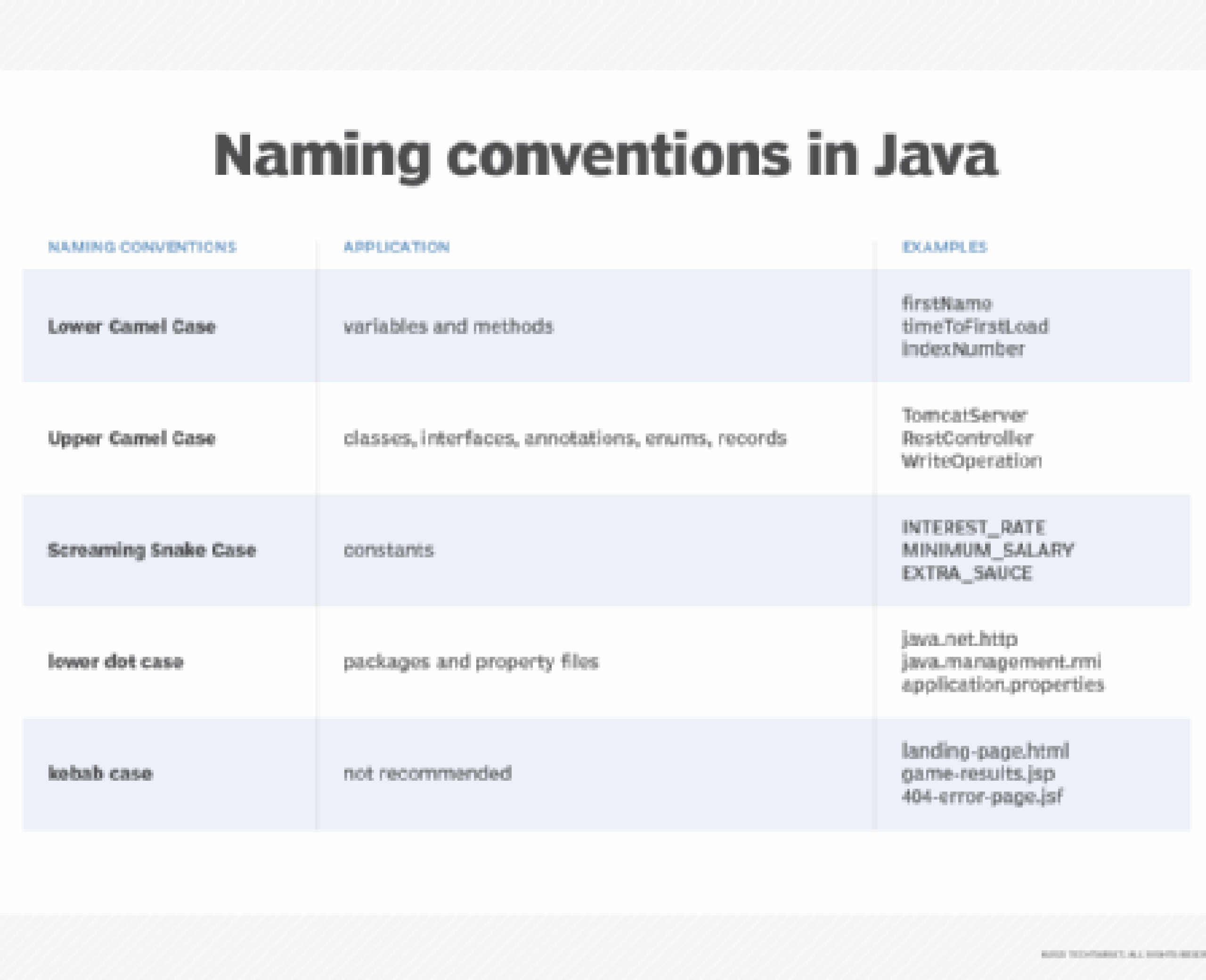
- PEP-8 is the official Python Style Guide
    - Follow this if other people will see your code (industry standard)
- You can also develop your own style (like myself)
- If you want to follow a common casing, choose one of these:
    - **snake_case**: all lowercase, spaces are replaced with an underscore (PEP-8 uses this for *nearly everything*)
    - **camelCase**: no spaces, first letter of every word except the start is captialized (never used in Python)
    - **PascalCase**: camel case, but every word is capitalized (Objects)

Formatting continued

Some languages use multiple cases. My personal style:

- snake_case: Variables
- camelCase: Functions
- PascalCase: Objects

I don't use Java... don't be getting any ideas.



Naming conventions in Java

NAMING CONVENTIONS	APPLICATION	EXAMPLES
Lower Camel Case	variables and methods	firstName timeToFirstLoad indexNumber
Upper Camel Case	classes, interfaces, annotations, enums, records	TomcatServer RestController WriteOperation
Screaming Snake Case	constants	INTEREST_RATE MINIMUM_SALARY EXTRA_SAUCE
lower dot case	packages and property files	java.net.http java.management.rmi application.properties
kebab case	not recommended	landing-page.html game-results.jsp 404-error-page.jsf

Variables

```
eggs = 20     # Sets the variable named `eggs` equal to 20
print(eggs)   # Prints the value of `eggs` to the terminal
>>> 20
```

- Something that stores data
- You can think of a variable like a food container
  - Name of the variable = Container label
  - Value in variable = Food itself
  - Data type = Type of food that can be stored in the container
  - Ex. `msg = "Hi"` stores the value "Hi" in a variable named `msg`

Data Types

- Type of data that can be stored (type of food for a container)
- Lots of data types in Python:
  - `int` - Any whole number (including negatives)
  - `float` - Any decimal number (including negatives)
  - `bool` - Binary value (`True` or `False`)
  - `list` - Array of values
  - `str` - List of characters
  - `dict` - List of key-value pairs
  - Others include: `range`, `bytes`, `tuple`, `set`, `None`

Examples of data types

- `int` - `10`, `23`, `-40`, `-96`, `1502089109740`, `-129379379`
- `float` - `10.0`, `-1532.32523409`, `-302.`, `3.141592`
- `bool` - `True`, `False`, `1`, `0`
- `list` - `[1, 5, False, True, -23940.212, -129423, 364, -0.15246]`
- `str` - `"Hello"`, `'a'`, `"Python loves memory"`, `"42.7837"`
- `dict` - `{"apples": 10, "oranges": 7, "bananas": 4, "grapes": 26}`

Why data types are important

- Python is a loosely typed language
    - Data types are implicitly set and sometimes implicitly casted
    - Ex. `eggs = 20` automatically assigns `eggs` the data type of `int`
- If data is used with the wrong data type, errors can be thrown
    - Ex. `10 == "10"` will return `False` because integers are different from strings
- Type casting changes the data type of a value
    - Type cast by returning a value with a constructor
    - Ex. `10 == int("10")` will return `True` because they have the same value after the type cast

Basic Math Operations

```
eggs = 20          # Sets the variable `eggs` to equal 20
eggs = eggs + 5    # Sets `eggs` to the current value of `eggs` and 5
print(eggs)        # Prints the value of `eggs` to the terminal
print(eggs % 5)    # Prints the remainder of `eggs` / 5
>>> 25
>>> 0
```

- Some basic operators include:
  - Addition, subtraction, multiplication, division, remainder (+-*/%)
  - Exponents, integer (floor) division (**, //)
- More operations can be done with the `math` module

Using Multiple Operations

```
apples = 10
oranges = 7
bananas = 4
fruit = apples + oranges + bananas
print(fruit)
>>> 21
```

```
num_fruits = 3
avg_fruits = (apples + oranges + bananas) / num_fruits
print(avg_fruits)
>>> 7.0
```

- PEMDAS rules apply

Compound Operators

```
eggs = 20
eggs += 5
eggs /= 5
print(eggs)
>>> 5.0
```

- Fast way to modify stored values
- Uses the operator appended with `=`
  - Ex. `biscuits *= 7` multiplies 7 to the variable `biscuits`
  - Ex. `fruits -= 10` subtracts 10 from the variable `fruits`
- Further operations can be used in the same line

Control Flow

- Doing different things under certain conditions
  - Compare values to determine a condition
  - Do something based on that condition
  - Ex. If a person has nut allergy, use an alternative ingredient
- Doing something repeatedly
  - Set the conditions of the repetition
  - Do something based on that condition
  - Ex. Start timer when chicken reaches 160F internally
  - Ex. Lightly butter all 12 biscuits

Comparison Operators

```
print(42 > 2**8)
>>> False
```

- Returns if a statement is `True` or `False`
- Operators:
  - `==`, `!=` - Equal to, Not equal to
  - `<`, `>` - Less than, Greater than
  - `<=` - Less than or equal to
  - `>=` - Greater than or equal to

Logical Operators

```
print(42 > 2**8 or 42 < 700)
print(not 42 >= 2**5)
>>> True
>>> False
```

- Returns if all statements are `True` and/or `False`
- Operators:
  - `and` - If at least one value is `False`, returns `False`
  - `or` - If at least one value is `True`, returns `True`
  - `not` - If value is `True`, return `False` (and vice versa)

`if` Statement

```python
customers = 3
if customers > 1:
    print("Wiping tables")
elif customers == 10:
    print("Restaurant full")
else:
    print("Waiting for customers")
```

- `if` - Runs the first codeblock if the condition is `True`
- `elif` - Runs the second codeblock if the first condition is `False` and second condition is `True`
- `else` - Runs the last codeblock if neither conditions are `True`

`for` Loop

```
result = 0
numbers = [67, -236, -112, 445, 14]
for num in numbers:
    result += num
    print(result)
>>> 67
>>> -169
>>> -281
>>> 164
>>> 178
```

- `num` is the value of the object at an internally tracked index
- `numbers` can be any iterable object (ie. `range`, `list`, etc.)

`while` Loop

```
num = 0
run = True
while run == True:
    num += 1
    if num > 100:
        run = False
    print(num)
>>> 1
>>> 2
>>> ...
>>> 101
```

- Checks if the condition is `True` before running the code block
- Once the condition returns `False`, the block will be skipped

Going further with loops

```
for num in range(10000):
    if num % 2 == 0: continue
    if num == 1000: break
    print(num)
>>> 1
>>> 3
>>> ...
>>> 9997
>>> 9999
```

- Both keywords can be used with either `for` or `while` loops
- `break` - Exits the loop at the executed line
- `continue` - Stops the code and starts again at the next iteration

Functions

```
# Adds 2 numbers together and says "Hello!"
def addHello(a, b):
    print(a + b)
    print("Hello!")

addHello(3, 5)
>>> 8
>>> Hello!
```

- Predefined block of code that can be called multiple times
- Allows for modularity, reuse of code, and clearer code
- Can optionally have parameters (do different stuff based on input) and return values (output data that can be reused in later code)

Function Structure Pt. 1

```python
def addHello(a, b):
    print(a + b)
    print("Hello!")
# Return to global scope code...
```

- `def` - Start a function definition
- `addHello` - Name of the function
- `(a, b)` - Parameters of the function (optional)
- Function scope - Code to run when function is called
    - Function scope ends when the indent is inline with `def` or if it returns a value

Function Structure Pt. 2

```python
def func1():                # No parameters or return type
def func2(a):               # One parameter
def func3(a, b):            # Two parameters
def func4(a = 7, b = 25):   # Two parameters, both with default values
def func5(a, b = 25):       # Two parameters, one has a default value
def func6(a: int, b: int):  # Two parameters that only allows integer values
def func7(a, b) -> int:     # Two parameters and a return type
```

- Parameter - A variable used internally by the function
  - Values can be passed through a parameter to change the output
- Return - Outputs a value and ends the function
  - Typically returned into a variable or `if` statement

Function Parameters

```
def doubleIt(num):  # Define `num` as a variable
    num = num * 2     # Set the value of `num` to `num * 2`
    print(num)        # Print the value of `num` to the terminal

doubleIt(6)         # Call the function while passing in the value 6
print(num)          # Attempt to print the value of `num`
>>> 12
>>> NameError
```

- `value` is used internally by the function (as a normal variable)
- `value` does not exist in the global nor previous scopes
  - You will get an error if you try to use a function's variable outside of its scope if it is not globally accessible

Function Returns

```
def doubleIt(value):
    return value * 2

num = doubleIt(6)
print(num)
>>> 12
```

- `return` will output data as a value that can be:
    - Stored in a variable
    - Used in another function
    - Output to the console

Function Scope

```
def doubleIt(value):
    doubled = value * 2
    return doubled
print(doubleIt(7))
print(doubled)
>>> 14
>>> NameError
```

- Variables created inside functions can only be accessed inside the function (unless specified with `global`)
  - `doubled` variable gets created inside the function
  - `print(doubled)` raises `NameError` since it doesn't exist globally