TechForNonTechies

Prerequisites

- No background in programming required
- No software needs to be installed. All coding is done using repl.it.
- Programming will be done in Python

Goals

- A whistle stop tour of the life a program
- Give the absolute basics of programming
- Each class broken up into two sections:
  - Background
  - Coding tutorial

- At the end you should be able to call an API and do something with the response
- Teach programming concepts while teaching standard practices within Mastercard

Week 1

Variables

Variables are carriers of data values labeled by a descriptive name. They store relevant information that is then used by a program to perform computations or output it to the user. All variables have a data type and a value. A data type is the kind of data that the variable is storing, and the value is the actual data.

Declaring a variable

```
age = 23
name = "Sam"
```

Primitives

Basic building blocks of all applications. You can create more complex data types (objects) out of these 4

primitives

Integer

They are positive or negative whole numbers with no decimal point

```
totalStudents = 100
temperature = -7
```

Boolean

The boolean data type is either True or False

```
isNightTime = False
isDayTime = True
```

Float

They represent real numbers and are written with a decimal point dividing the integer and the fractional parts.

```
weight = 17.34
average = -34.0
```

String

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in single or double quotes.

```
name = "John"
lastName = 'Doe'
```

Collections

List

Sequence of values encapsulated in a single variable. Are created by putting comma separated values insisde square brackets. Elements inside a list can be of different data types and can change.

```
list1 = ['physics', 'chemistry', 1997, 2000]
list2 = [1, 2, 3, 4, 5 ]
list3 = ["a", "b", "c", "d"]
```

To retreive a particular element of a list, we must enclose the index of the list we want in square brackets. Note: Indexes in python go from 0 to n-1, with n being the size of the list.

```
firstElement = list1[0] #gets the first value of list1 ('physics')
lastElement = list1[3] #gets the last value of list1 (2000)
```

We can also modify the values of the list using a similar approach.

```
list1[0]= 'biology' #Now list1 looks like this ['biology', 'chemistry', 1997, 2000]
list1[3] = 3000 #Now the list1 looks like this ['biology', 'chemistry', 1997, 3000]
```

Dictionary

Are squences of key-value pair, where every key in the sequence is unique, but values may not be. More than one entry per key is not allowed. Keys can be of any type but can't change. Values can change. Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.

```
dict = {'Name': 'Sam', 'Age': 23, 'isLefty': False}
```

Using Dictionaries:

Modifying Dictionaries:

Tuple

Same as lists but with the main difference that the elemens inside of it can't change. Created by putting different comma-separated values, and can optionally be wrapped inside parenthesis

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup3 = "a", "b", "c", "d"
```

Using Tuples:

Set

TODO: Add set section

Operators

Mathematical

Addition (+)

Subtraction (-)

Multiplication (*)

Division (/)

Exponentiation (**)

Modular (%)

Exercises

See appendix

Quiz

- Simple quizes to reinforce the information presented
- Quiz before and after each session?

Week 2

Recap and looking ahead

In week 1 we were introduced to primitive data types, collections, and mathematical operators.

This week we will explore two other types of operators and how we can use them to control the flow of our program. We will then take a look at a special control specifically designed for collections.

The topics covered this week include:

- Comparison Operators

- Logical Operators
- Control Structures
- Conditionals
- Loops

Operators

Review: Arithmetic Operators

Arithmetic operators typically take two numeric operands and return a result of the specified operation.

Example:

```
val = 5 + 7 # operator is +; left operand is 5; right operand is 7
```

One special scenario to consider is when using the addition operator to concatenate two strings.

Example:

```
result = "Hello, " + "world"

print(result)   # prints "Hello, world"
```

Comparison Operators

Comparison operators, sometimes called relational operators, also take two operands which are often numeric. The main difference from arithmetic operators is that these operators test the relationship between (compare) the operands and return a `boolean` value, either `True` or `False`

These operators enable our programs to make decisions based on certain conditions for which they can test.

Example: To enforce parental controls, a program might check that the current time is earlier than 9pm before granting a user (the parents' child) access, or;

To authorize an incoming API request, we need to validate that the request token was issued by Mastercard.

Similar to the `+` operator used to concatenate two strings, some comparison operators can be used to compare strings as well as numbers.

Operator	Description	Example

`==`	Is the left operand equal to the right operand?	`1 == 2` returns `False`
		`"hello" == "hello"` returns `True`
		`"1" == 1` returns `False`

`!=`	Is the left operand *not* equal to the right operand?	`1 != 2` returns `True` `"hello" != "hello"` returns `False`
`>`	Is the left operand greater than the right operand?	`5 > 5` returns `False`
`>=`	Is the left operand greater than or equal to the right operand?	`5 >= 5` returns `True`
`<`	Is the left operand less than the right operand?	`5 < 5` returns `False`
`<=`	Is the left operand less than or equal to the right operand?	`5 <= 5` returns `True`

There are two special operators used to test equality in a very specific way. We'll return to these next week.

Operator	Description	Example

`is`	Is the left operand the same object as the right operand?	`p1 = {"name":"Bret"}`
		`p2 = {"name":"Bret"}`
		`p1 == p2` returns `True`
		`p1 is p2` returns `False`

`is not`	Is the left operand *not* the same object as the right operand?	`p1 != p2` returns `False` `p1 is not p2` returns `True`

Logical Operators

Logical operators, sometimes called boolean operators, take either one or two `boolean` operands and return a `boolean` result.

These operators allow us to make decisions in our program based on *multiple* conditions which may change over time. A logical operator can take the result of a comparison and combine it with the result of another comparison to return a new result.

Operator	Description	Example

`and`	Are the left operand *and* the right operand both `True`?	`True and True` returns `True`
		`True and False` returns `False`
		`False and False` returns `False`

`or`	Is either the left operand *or* the right operand `True`?	`True or True` returns `True`
		`True or False` returns `True`
		`False or False` returns `False`

`not`	Is the inverse of the operand `True`?	`not True` returns `False` `not False` returns `True`

Control Structures

Control structures determine the execution flow of your code and enable a program to change its behavior based on its current state. State can change due to any number of reasons including user input, time of day, network avaialability, and navigation. This means your program can respond to state changes and prevent unexpected behavior.

Sequence

By default, all of your code is executed sequentially, with each line executing in the exact order in which it's written. A program written this way will always execute the same code no matter what happens. This works fine for very simple programs with predetermined and guaranteed state.

```
# A simple program which will always work when executed sequentially
x = int(input('Enter a number: '))
y = int(input('Enter another number: '))

print(f'Sum of {x} and {y} is {x + y}')
```

Selection

If your program needs to behave differently dependending on its current state, you can introduce **conditional statements** to allow your your program to *select* flows of exeuction for different states.

For more complex programs (99.9999% of all software), you will need to ensure that the program behaves correctly by evaluating its current state and choosing what to do next.

`if` Statements

In Python, selections are written in the form of `if-elif-else` statements.

`if` statements are given a `conditional expression` which evaluates to either `True` or `False`. If the expression evaluates to `True` the following code block is executed. If the expression evaluates to `False` the block is skipped.

```
# Only show the time if the user wants to know

show_time = input('Do you want to know the current time (Y/n)? ')

if show_time == 'Y':
    print(datetime.now())   # This is skipped if the user enters 'n'
```

`elif` provides a sort of fallback in case the preceeding `if` evaluates to `False`. This is useful when there are multiple possible outcomes you want to test for. The block following `elif` will executing *only* if preceding `if` and `elif` statements evaluate to `False` and this `elif` statement evaluates to `True`

```
# Normalize your user input to prevent negative side-effects

num = int(input('Give me a number between 0 and 100: '))

if num < 0:
    num = 0
    print('Your negative number was rounded to 0')
elif num > 100:
    num = 100
    print('Your number was rounded to 100')

use_number(num)
```

`else` provides a fallback that is guaranteed to run if all preceeding `if` and `elif` statements evaluate `False`. This is useful if you can't predict all possible outcomes and want to provide a default behavior.

```
# Enable your program to behave differently according to its input

result = None
x = int(input('Give me a number: '))
y = int(input('Give me another number: '))
op = input('What do you want to do (+, -, *, /, **): ')

if op == '+':
    result = x + y
elif op == '-':
    result = x - y
elif op == '*':
    result = x * y
elif op == '/':
    result = x / y
elif op == '**':
    result = x ** y
else:
    print("I don't know what that means...")

if result is not None:
    print(f'{x} {op} {y} = {result}')
```

More examples

Test whether a number falls between two other numbers

```
lock_start_hour = 21              #Restrict access starting at 9pm (21:00)
lock_end_hour = 6                 #Remove restriction at 6am (06:00)
current_hour = datetime.now().hour   #Get the current hour

if current_hour >= lock_end_hour and current_hour < lock_start_hour:
    access_allowed() # Access allowed outside of restricted hours
else:
    access_refused() # Otherwise access is refused

# Can you spot the bug in this code?
```

Iteration

Often times we want to execute the same block of code multiple times. Maybe we have a collection of values

that we want to display to the user or we want the user to input an undetermined number of values from the terminal.

When we need to iterate through a collection of items and perform some task, a `for` loop is usually the best choice. If the number of loops required is unknown, we can use a `while` loop provided with a `conditional expression`.

`for` Statements

Given a collection of items, a `for` loop enables a program to perform some task on each item in the collection individually.

```
# Generate average spending of users

users = get_all_users() # Returns a list of all users
total_spent = [] # A collection to hold each user's total ($) spent

for user in users:  # For each user
    total_spent.append(user.total_spent)  # Record each user's total ($) spent

med_spent = median(total_spent)

print(f'The median amount spent for all users is {med_spent}')
```

`while` Statements

A `while` loop continuously repeats a block of statements as long as a given conditional expression evaluates to `True`. This is fundamentally different from a `for` loop, which iterates over the items contained within a given collection.

```
# Generate a report from recent transactions

trans = []
value = int(input('Enter your recent transaction amounts (-1 to exit):'))

while value != -1:
    trans.append(value)

    value = int(input())

trans_total = sum(trans)
trans_median = median(trans)
trans_mean = mean(trans)

print(f'Total: {sum(trans)}, median: {median(trans)}, mean: {mean(trans)}')
```

Exercises

See appendix

Quiz

TODO: Come up with quiz questions

Week 3

Week 2 enabled us to write more complex and useful programs by using control structures such as selections (`if-elif-else`) and iterations (`for` and `while`). We can now evaluate the current state of our program and make decisions by testing their conditions.

Review Quiz

1. Translate the following statement into code: Given variables x and y, if x is greater than y, print their sum, but if x is less than or equal to y, print their difference.

```
if x > y:
    print(x + y)
elif x <= y:
    print(x - y)    # Technically, difference should be an absolute value

# or, more concisely

if x > y:
    print(x + y)
else:
    print(x - y)
```

1. Translate the following statement into code: Input a list of users' names until an empty return and then print out all of the names that were entered.

```
users = []
name = input()

while name != "":
    users.append(name)

    name = input()

for name in users:
    print(name)
```

1. Translate the following statement into code: Given variables x and y, if x plus y is greater than 0 but less than 100, print the sum. Otherwise print "Out of bounds."

```

```

Classes, Objects, and Functions

A class can be thought of as a blueprint that defines state and behavior of a particular *thing*. In the real world, you often have many objects of the same kind. For example, your bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle object is an *instance* of the *class* of *objects* known as bicycles. Bicycles have some state (current gear, current cadence, two wheels) and behavior (change gears, brake) in common. However, each bicycle's state is independent of and can be different from that of other bicycles.

While you may not know exactly what a class or object is in programming terms, you've actually been using

both for the last 2 weeks! An object is an instance of a complex data type which can hold many other different types of data, and an object is an instance of that complex data type. Sound familiar? All of the collections we worked with in weeks 1 and 2 are objects!

Define a Class

A class is defined with 3 components in mind:

- Name
- Attributes
- Functions

```

```

- Objects
    - Functions

Running software

- Pivotal Cloud Foundry
    - Sample push application
    - Pipelines
- Jenkins
- CI/CD

Week 4

How does the internet work?

- Show the basics of an HTTP request?
- What are microservices?
- What is an API?

Exercises

Call an API

- Make a call to the DarkSky API to retrieve the weather for Dublin

- Do something with the result (e.g. get the max wind speed for the day)
- **Advanced**: call a Mastercard API
    - Requires using the OAuth signer

Summary of learning

Goals

- Graduates should be able to identify terminology
    - API
    - JSON
    - Keys
    - XML
    - Git
    - Maven, Gradle
    - Microservices
- Relate what they learned in the course to their daily work lives
- Capstone project
    - DarkSky API integration

Tools

- Repl.it
- Jupyter?
- Documentation!

Exercises

Week 1

Exercise 1: Variables

Write a python program that creates a list with five elements your name, last name, age, height (in), and whether you are lefty (True or False), then create three variables and assign it the values of the firs, middle, and last element of the list respectively, and lastly prints to screen these three variables.

```
list1 = ["Hello", "World", 5, 3.56, False]
first = list1[0]
middle = list1[2]
last = list1[4]
print(f"{first} {middle} {last}")  #prints: Hello 5 False
```

Exercise 2: Say hello!

```
print('Please tell me your name')
name = input()
print(f'Hello {name}')
```

Week 2

Exercise 1: Authentication

Before we grant a user access to our system, they need to sign in to an existing account using the correct username and password combination. This is a very basic form of authentication.

Write a program which allows a user to input their username and password. Verify that the username exists and that the entered password matches the password associated with this username.

Acceptance Criteria

1. If the username entered does not exist, print an error message.
2. If the username does exist, allow the user to enter the password.
3. If the password entered does not match the correct password, print an error message.
4. If the password does match, print a success message.

Extra credit:

1. Allow the user to enter their password up to *three* times before quitting the program.
2. Allow the user to create an account if it does not already exist.