CUHackit Python Workshop

Guided Tour Through My Favorite Python-isms

Alex Day

Ph.D. Student
School of Computing

shorturl.at/lrFH2

January 29, 2022

CLEMSON
College of ENGINEERING,
COMPUTING AND APPLIED SCIENCES

Overview

Overview

Alex Day

- B.S. Computer Science Clarion in Pennsylvania
- Ph.D. Student under Dr. Ioannis Karamouzas
- Working on Social Robot Navigation in the Motion Planning Lab in Charleston
- Programming in Python for the last 7 years

Audience

- What year are you?
- How comfortable are you with programming in general, any language?
- How comfortable are you with python?

Audience

- What year are you?
- How comfortable are you with programming in general, any language?
- How comfortable are you with python?

Audience

- What year are you?
- How comfortable are you with programming in general, any language?
- How comfortable are you with python?

Python

- First released in 1991 by Guido van Rossum (BDFL)
- Python 3 was released in 2008
- Python Enhancement Proposals (PEPs)
- Scripting language popular in machine learning and for rapid prototyping
- Massive repository of third party libraries on PyPi

Python

Example

```
name = input("What is your name?")

if name.lower() == "alex":
    print("Hi!")
else:
    for _ in range(5):
        print("GO AWAY I DON'T KNOW YOU")
```

Input and Output

Example

```
print("Hello world!")
user_input = input("> ")
print(user_input)
```

Conditionals

Example

```python
import random

number_to_guess = random.randint(0, 100)
guess = int(input("Please enter your guess... "))

if guess < number_to_guess:
    print("too small")
elif guess > number_to_guess:
    print("too big")
else:
    print("just right")
```

Iteration (While)

Example

```python
import random

number_to_guess = random.randint(0, 100)
print(number_to_guess)

while True:
    guess = int(input("Please enter your guess... "))

    if guess < number_to_guess:
        print("too small")
    elif guess > number_to_guess:
        print("too big")
    else:
        print("just right!")
        break
```

Iteration (For)

Example

```python
import random

number_to_guess = random.randint(0, 100)
print(number_to_guess)

for i in range(5):
    guess = int(input(f"Please enter your guess ({i}/5)... "))

    if guess < number_to_guess:
        print("too small")
    elif guess > number_to_guess:
        print("too big")
    else:
        print("just right!")
        break
else:
    print("You'll get em next time")
```

Functions

Example

```python
def fib(n, a=0, b=1):
    """Return the first n numbers of the Fibonacci sequence

    Keyword arguments:
    n -- length of sequence
    a -- first number
    b -- second number
    """
    fib = []
    for i in range(n):
        fib.append(a)
        a, b = b, a + b

    return fib

print(fib(5))
# >>> [0, 1, 1, 2, 3]
```

Classes

Example

```python
from random import choice

class Dog:
    def __init__(self, name, breed):
        self._name = name
        self._breed = breed

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name: str):
        self._name = name

    def bark(self):
        print("WOOF")

    @staticmethod
    def make_random_dog():
        names = ["Frido", "Spot", "Kansas", "Lucky"]
        breeds = ["German Shepard", "Border Collie", "Golden Doodle", "Labrador"]
        return Dog(choice(names), choice(breeds))
```

Types?

- Python is duck typed
    - If it walks like a `float` and looks like a `float` then it's probably a `float`

Example

```
word = 'Hello'
word = 7
word = 3.63
word = {}
```

- The lack of type safety has pros and cons
- `typing` and `mypy` aim to make python statically typed

Typing in Python

Example

```python
from typing import List

def fib(n: int, a: int=0, b: int=0) -> List[int]:
    """Return the first n numbers of the Fibonacci sequence

    Keyword arguments:
    n -- length of sequence
    a -- first number
    b -- second number
    """
    fib = [] # type: List[int]
    for i in range(n):
        fib.append(a)
        a, b = b, a + b

    return fib

print(fib(5))
#>>> [0, 1, 1, 2, 3]
```

Typing in Python

Example

```
mypy functions_typed.py
# Success: no issues found in 1 source file
```

Overview

Lists

- Lists are reference types, similar to `ArrayLists` in Java or `Vectors` in C++
- Unlike other languages lists do not have a specific type

Example

```
numbers = [1, 2, 3, 4, 5, 6]

# Indexing
print(numbers[1])
>>> 2

# Slicing
print(numbers[1:])
>>> [2, 3, 4, 5, 6]

wacky = [1, 1.5, "one point five", {}]
```

List Comprehension

- Just a syntactical nicity
- Intuitive to read even for a non-programmer

Example

```
[expression for item in list]
```

Generation of Lists

- Allows terse generation of complicated risk

Example

```
squares = []
for x in range(10):
    squares.append(x**2)

squares = [x**2 for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Filtering of Lists

- Allows you to filter lists using a boolean function

Example

```
def is_prime(n: int) -> bool:
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

primes = []
for x in range(100):
    if is_prime(x):
        primes.append(x)

primes = [x for x in range(100) if is_prime(x)]
# [0, 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Cool List Comprehensions

- There are also some pretty neat things you can do

Example

```
mat = [[1, 2, 3], [4, 5, 6]]
flat = [x for row in mat for x in row]
# [1, 2, 3, 4, 5, 6]


sent = "The quick brown fox jumps over the lazy dog"
num_vowels = len([letter for letter in sent if letter in "aeiou"])
# 11
```

Overview

Dunder Methods

- Dunder (**d**ouble-**under**score) methods are special methods that can be implemented in objects
- Similar to the concept of `toString` in Java
- These methods allow special behavior such as:
  1. `__repr__(self)` controls how the object is shown as a string
  2. `__getitem__(self, idx)` allows you to implement indexing on your object
  3. `__contains__(self, item)` allows you to implement the `in` keyword functionality
- exhaustive list in the Python documentation

`__repr__`

- Meant to return a debugging-quality string representing the object
- Similar to `__str__`
  - `__str__` is for a user-facing, ambiguous, incomplete string representation
  - `__repr__` is for a developer-facing, unambiguous, complete string representation

Example

```python
class Dog:
    def __init__(self, name: str, breed: str, collar_color: str):
        self._name = name # type: str
        self._breed = breed # type: str
        self._collar_color = collar_color

    def __repr__(self) -> str:
        return f"name: {self._name}, breed: {self._breed}, collar_color: {self._collar_color}"

    def __str__(self) -> str:
        return "Your dog, {self._name} is a {self._breed}"
```

`__lt__`, `__gt__`, `__eq__`, etc.

- Rich comparison operators which work as you would expect
- The signatures all take the same form of `__lt__(self, other)`
- These functions *should* return either `True` or `False`

Example

```python
class ChargingBrick:
    def __init__(self, volts: float, amps: float):
        self._volts = volts # type: float
        self._amps = amps # type: float

    def __lt__(self, other: "ChargingBrick") -> bool:
        return self._volts * self._amps < other._volts * other._amps

    def __gt__(self, other: "ChargingBrick") -> bool:
        return self._volts * self._amps > other._volts * other._amps

    def __eq__(self, other: "ChargingBrick") -> bool:
        return not self > other and not self < other
```

`__getitem__`, `__setitem__`, and `__delitem__`

- Allows indexing and indexing related operations like slicing
- Primarily implemented on custom container objects like dictionaries or lists
- `__getitem__(self, key)`
  - Returns a reference to the object at `key`
  - Have to special implement the negative indexes
- `__setitem__(self, key, value)`
  - Allows the user to insert a `value` into a `key`
  - Should only be implemented if you can also remove from the container
- `__delitem__(self, key)`
  - Allows the user to delete an item at a specific `key`

__getitem__, __setitem__, and __delitem__

Example

```
from typing import Any

class Node:
    def __init__(self, datum: Any=None):
        self.datum = None # type: Any
        self.next = None # type: Node
```

__getitem__, __setitem__, and __delitem__

Example

```
class LinkedList:
    def __init__(self):
        self.root = Node()

    def append(self, value: Any):
        tmp = self.root
        while tmp.next is not None:
            tmp = tmp.next

        new = Node(value)
        tmp.next = new

    def __getitem__(self, key: int) -> Any:
        tmp = self.root
        i = 0
        while tmp.next is not None:
            i += 1
            tmp = tmp.next

            if i == key:
                return tmp.datum

        raise IndexError()
```

__getitem__, __setitem__, and __delitem__

Example

```
def __delitem__(self, key: int) -> Node:
    tmp = self.root
    i = 0
    while tmp.next is not None:
        i += 1
        tmp = tmp.next

        if i == key:
            if tmp.next.next is not None:
                tmp.next = tmp.next.next
            else:
                tmp.next = None

    raise IndexError()
```

Overview

Walrus Operator

- Another nicity that allows more terse code
- Normally the item assignment action doesn't return any value
- Using the walrus operator (`:=`) causes the assignment to return that variable

Walrus Operator

Example

```
# Simple Assignment
num = 15

print(num)
# >>> 15

print(num = 15)
# >>> TypeError

# Walrus
print(num := 15)
# >>> 15
```

Walrus Operator

Example

```
# Get and test user input
inp = input("> ")
if inp.lower() in ["y", "yes"]:
    print("you said yes!")

if (inp := input("> ")).lower() in ["y", "yes"]:
    print("You said yes!")
```

The End