6.00.1.x - Week 2 - Simple Programs

September 17, 2020

1 Video 1: So far

Reviewing strings:

- A sequence of case sensitive characters.
- Can compare them with ==, <, >, etc.
- len() is a function used to retrieve their **length**.
- Square brackets are used to perform **indexing** into a string to get the value at a certain index/position.
- Can **slice** strings using [start:stop:step].
- Strings are "**inmutable**" - cannot be modified.

[4]:
```
s = 'hello'
print(s)
type(s)
s[0]
```

```
hello
```

[4]:
```
'h'
```

[5]:
```
s[0] = 'y'
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-5-88dd6fabac11> in <module>
----> 1 s[0] = 'y'

TypeError: 'str' object does not support item assignment
```

[7]:
```
s = 'y' + s[1:]
print(s)
```

```
yello
```

2 Video 2: Approximate Solutions

Observations

1. Step could be any small number.
    1. If too small, takes a long time to find square root.
    2. If too large, might skip over answer without getting close enough.
2. In general, will take x/step times through code to find solution.
3. We need a more efficient way to do this.

3 Video 3: Bisection Search

At each stage, reduce range of values to search by half. This ia a great example of a logarithmic time algorithm finding an answer very quickly. This idea will be further developed later on in the course.

This should work on any problem with **ordering property** - value of function being solved varies monotonically with input value.

4 Video 4: Floats and Fractions

How do float represent real numbers? It is useful to get a sense of how the machine actually stores the numbers.

Internally, the computer represents every number in binary, whether it's an integer or a float. The algorithm of how it does it is slightly complex and was shown using two examples. Follow those if interested in the actual method.

Some implications

- If there is no integer p such that x*(2^p) is a whole number, then internal representation is always an approximation.
- Suggest that testing equality of floats is not exact:
  - Use abs(x-y) < some small number, rather than x == y.
- Why does print(0.1) return 0.1, if not exact?
  - Because Python designers set it up this way to automatically round.

5 Video 5: Newton-Raphson

- General approximation algorithm to find roots of a polynomial in one variable.
- Newton showed that if g is an approximation to the root of p, then g - p(g)/p'(g) is a better approximation, where p' is a derivate of p.

- So far we've seen the following methods of generating guesses:
  - Exhaustive enumeration.
  - Bisection search.
  - Newton-Raphson (for root finding).

6 Video 6: Decomposition and Abstraction

Good programming should be measured by the amount of functionality: the ability to make computations easily.

In order to become better programmers we're gonna introduce the concept of a function. But before that, we need to introduce two concepts: decomposition and abstraction.

Abstraction: do not need to know how something works in order to be able to use it.

```
* Suppress details of method to compute something from use of that computation.
```

Decomposition: different devices work together to achieve an end goal.

```
* Break problem into different, self-contained, pieces.
```

This lecture, we will achieve decomposition with **functions**. In a few weeks, achieve decomposition with **classes**.

We will achieve abstraction with **function specifications** or **docstrings**.

7 Video 7: Introducing Functions

- Characteristics
  - Name
  - Parameters (0 or more).
  - Docstring (optional but recommended)
  - Body.
- How to write it?

[3]:
```
def is_even(i):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("hi")
    return i%2 == 0
is_even(3)
```

```
hi
```

[3]:
```
False
```

Between """ we have the specification (docstring) of the function. Indented is the "body" of the function.

8 Video 8: Calling Functions and Scope

- **Formal parameters** gets bound to the value of **actual parameter** when function is called.
- New **scope/frame/environment** created when enter a function.
- **Scope** is a mapping of names to objects.

When a variable is changed within a function, it is changed in the function's scope but not in the global scope. Anytime a function is invoken, I create a new frame. Once I'm done with the body of the function, the frame is erased, because I no longer need it.

Return vs Print

- Return only has meaning **inside** a function, while print can be used **outside** functions.
- Only **one** return executed inside a function, while print can be executed many times.
- Code inside function but after return statement are not executed.
- Return has a value associated with it, **given to function caller**, while print has a value associated with it **outputted** to the console.

Arguments can take on any type, even functions.

Scope

- Inside a function, **can access** a variable defined outside.
- Inside a function, **cannot modify** a variable defined outside.

9 Video 9: Keyword Arguments

You may use an argument that changes how the function works. It branches out the functionalities provided by the function itself.

Interesting functionality: One may invoke the function inputting the parameters in a different order. Example: def fun(a,b,c): return a * b + c z = fun(b = 7, a = 5, c = 2)

Another interesting tool is being able to define a default value for a parameter such that, in case it is not explictly passed as an input when invoking the function, the parameter will take specifically that value.

10 Video 10: Specification

A **contract** between the implementer of a function and the clients who will use it.

- **Assumptions**: conditions that must be met by clients of the function; typically constraints on values of parameters.
- **Guarantees**: conditions that must be met by function providing it has been called in manner consistent with assumptions.

11 String Methods

11.1 Activity 13 of 27

String methods used:

- replace
- count
- find
- index
- swapcase
- capitalize
- islower
- isupper
- upper

https://docs.python.org/3/library/stdtypes.html#string-methods

12 Video 11: Iteration vs Recursion

Recursion

- A way to design solutions to problems by divide-and-conquer or decrease-and-conquer.
- A programming technique where a function **calls itself**.
- The goal is to NOT have infinite recursion:
  - Must have 1 or more base cases that are easy to solve.
  - Must solve the same problem on some other input with the goal of simplifying the larger problem input.

There's two structures that need to be identified in a problem that is solved by recursion:

1. Recursive step: how to reduce problem to a **smaller/simpler** version of itself.
2. Base case: Reduce the problem until you reach a simple case that can be **solved directly**.

[1]:
```
# Multiplication between 2 numbers
def mult(a,b):
    # Base case
    if b == 1:
        return a
    # Recursive step
    else:
        return a + mult(a,b-1)

mult(4,5)
```

[1]: 20

[3]:
```
# Factorial of a number n
def fact(n):
    # Base case
    if n ==1:
        return 1
    # Recursive step
    else:
        return n*fact(n-1)

fact(6)
```

[3]: 720

- Each recursive call to a function creates its **own scope/environment**.
- **Bindings of variables** in a scope is not changed by recursive call.
- Flow of control passes back to **previous scope** once function call returns value.

13 Video 12: Inductive Reasoning

How do we know that our recursive code will work?

- Invoking the recursive function with an ever decreasing input parameter.
- Another tool: Mathematical Induction. To prove a statement indexed on integers is true for all values of n:
  - Prove it is true when n is smallest value (e.g. n=0 or n=1).
  - Then prove that if it is true for an arbitrary value of n, one can show that it must be true for n+1.

14 Video 13: Towers of Hanoi

Somewhat complex problem that can be solved "pretty easily" by implementing recursive functions.

15 Video 14: Fibonacci

Can have recursion with multiple base cases too!

To generate the Fibonacci sequence:

- Base cases:
  - Females(0) = 1
  - Females(1) = 1
- Recursive case:
  - Females(n) = Females(n-1)+Females(n-2)

[1]:
```
def fib(x):
    """assumes x an int >= 0
       returns Fibonacci of x"""
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)

print(fib(7))
```

```
21
```

16 Video 15: Recursion on non-numerics

Solving recursively whether a string is or not a palindrome.

[1]:
```
def isPalindrome(s):

    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans

    def isPal(s):
        if len(s) <= 1:
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1])

    return isPal(toChars(s))

print("")
print('Is eve a palindrome?')
print(isPalindrome('eve'))

print('')
print('Is able was I ere I saw Elba a palindrome?')
print(isPalindrome('Able was I, ere I saw Elba'))
```

```
Is eve a palindrome?
True

Is able was I ere I saw Elba a palindrome?
True
```

17 Video 16: Files

Module: a .py file containing a collection of Python definitions of statements. Example: circle.py.

How to use a file?

1. Import it by writing: import *circle*
   - In order to avoid using circle.function1(arg) everytime, I can instead use the following command: from circle import *.
2. Open it by using the command open:

   nameHandle = open('filename','w') / ('w' for write or 'r' for read).

   Then one might write inside it by invoking the method nameHandle.write(whatever i want to write). Finally one might close the file by invoking the method nameHandle.close()