UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Lecture #2: Programming Structures: Loops and Functions

September 16, 2019 http://cs88.org

Administrivia

- Everyone should be enrolled now
- iClickers: Start next week.

Computational Concepts Today

- Fundamentals of Python
- Conditional Statements
- Functions
- Lists
- Iteration

Data or Code? Abstraction!

Human-readable code (programming language)

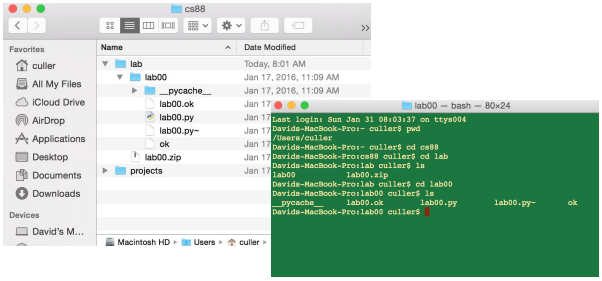
```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodename()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print ' %s [label="%s"' % (nodename, label),
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s"];' % ast[1]
        else:
            print '"];'
    else:
        print '"];'
        children = []
        for n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print ' %s -> {' % nodename,
        for name in children:
            print '%s' % name,
```

Machine-executable instructions (byte code)

Compiler or Interpreter
Here: Python

Code or GUI: More Abstraction!



- Big Idea: Layers of Abstraction
  - The GUI look and feel is built out of files, directories, system code, etc.

Let's talk Python

- Expression `3.1 * 2.6`
- Call expression `max(0, x)`
- Variables
- Assignment Statement `x = <expression>`
- Define Function: `def` <function name> `(`<argument list>`):`
- Control Statements:
  - if …
  - for …
  - while …
  - list comprehension

Conditional statement

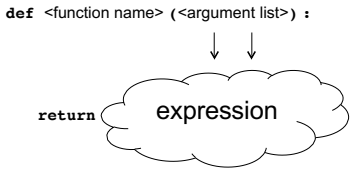
- **Do some statements, conditional on a *predicate* expression**

```
if <predicate>:
        <true statements>
else:
        <false statements>
```

- **Example:**

```
if (temperature>37.2):
        print("fever!")
else:
        print("no fever")
```


Defining Functions



- **Abstracts an expression or set of statements to apply to lots of instances of the problem**
- **A function should *do one thing well***

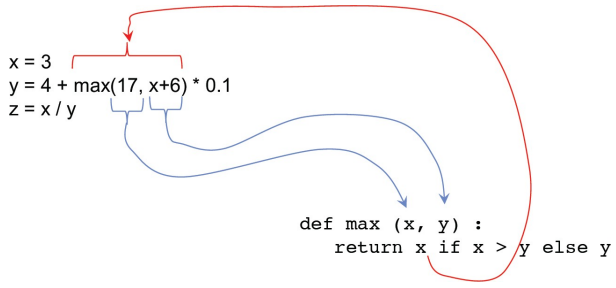
Functions: Calling and Returning Results

```
def my_function(number):
        print(argument)
        statements
        return number

data = my_function(1)

# data becomes whatever value is, returned.
# In this case data holds the value of number.
# 1 is an _argument_ to my_function,
# the argument number will be 1.
```


Functions: Example



How to write a good Function

- **Give a descriptive name**
  - Function names should be lowercase. If necessary, separate words by underscores to improve readability. Names are extremely suggestive!
- **Chose meaningful parameter names**
  - Again, names are extremely suggestive.
- **Write the docstring to explain *what* it does**
  - What does the function return? What are corner cases for parameters?
- **Write doctest to show what it should do**
  - **Before** you write the implementation.

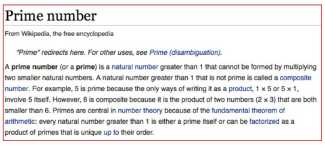
Python Style Guide: https://www.python.org/dev/peps/pep-0008/

Example: Prime Numbers

```
def prime(n):
    """Return whether n is a prime number.

    >>> prime(2)
    True
    >>> prime(3)
    True
    >>> prime(4)
    False
    """

    return "figure this out"
```


Prime number

From Wikipedia, the free encyclopedia

"Prime" redirects here. For other uses, see Prime (disambiguation).

A **prime number** (or a **prime**) is a natural number greater than 1 that cannot be formed by multiplying two smaller natural numbers. A natural number greater than 1 that is not prime is called a composite number. For example, 5 is prime because the only ways of writing it as a product, 1 × 5 or 5 × 1, involve 5 itself. However, 6 is composite because it is the product of two numbers (2 × 3) that are both smaller than 6. Primes are central in number theory because of the fundamental theorem of arithmetic: every natural number greater than 1 is either a prime itself or can be factorized as a product of primes that is unique up to their order.

Why do we have prime numbers?
https://www.youtube.com/watch?v=e4kevnq2vPI&t=72s&index=6&list=PL17CtGMLr0Xz3vNK31TG7mJlzmF78vsFO

`list` – A data structure for iteration

- **A list is a collection of items in a single group.**
- **They can hold just about anything.**

```
my_list = [1, 2, 3]

my_courses = ['CS88', 'DATA8', 'MATH1A']

len(my_courses) == 3 # len returns the
length

print(my_courses[0]) # prints CS88
```


`for` statement – iteration control

- **Repeat a block of statements for a structured sequence of variable bindings**

```
<initialization statements>
for <variables> in <sequence expression>:
  <body statements>

<rest of the program>
```

```
def cum_OR(lst):
    """Return cumulative OR of entries in lst.
    >>> cum_OR([True, False])
    True
    >>> cum_OR([False, False])
    False
    """
    co = False
    for item in lst:
            co = co or item
    return co
```


`while` statement – iteration control

- **Repeat a block of statements until a predicate expression is satisfied**

```
<initialization statements>
while <predicate expression>:
   <body statements>

<rest of the program>
```

```
def first_primes(k):
    """ Return the first k primes.
    """
    primes = []
    num = 2
    while len(primes) < k :
        if prime(num):
            primes = primes + [num]
        num = num + 1
    return primes
```


Data-driven iteration

- **describe an expression to perform on each item in a sequence**
- **let the data dictate the control**

```
[ <expr with loop var> for <loop var> in <sequence expr > ]
```

```
def dividers(n):
    """Return list of whether numbers greater than 1 that divide n.

    >>> dividers(6)
    [True, True]
    >>> dividers(9)
    [False, True, False]
    """
    return [divides(n,i) for i in range(2,(n//2)+1) ]
```