Session 3 recap

Strings, tuples and lists

Comparison

Type	Type of elements	Examples	Mutable
str	characters	"", "abc", "2"	No
tuple	any type	(), (3,), ("abc", 3)	No
list	any type	[], [3], ["abc", 3]	Yes

Common operations

seq[i] returns the i^{th} element in the sequence.
len(seq) returns the length of the sequence.
seq1 + seq2 returns the concatenation of the two sequences.
n * seq returns a sequence that repeats seq n times.
seq[start:end] returns a slice of the sequence.
e in seq is True if e is contained in the sequence and False otherwise.
e not in seq is True if e is not in the sequence and False otherwise.
for e in seq iterates over the elements of the sequence.

Dictionaries

- A **dictionary** is a set of **key:value** pairs.
- Pairs are separated by commas and enclosed within curly brackets {}
- Pairs are unordered, we access dictionaries using keys.

Common operations

`len(d)` returns the number of items in `d`.

`d.keys()` returns a list containing the keys in `d`.

`d.values()` returns a list containing the values in `d`.

`k in d` returns `True` if key `k` is in `d`.

`d[k]` returns the item in `d` with key `k`.

`d.get(k, v)` returns `d[k]` if `k` is in `d`, and `v` otherwise.

`d[k] = v` associates the value `v` with the key `k in d`. If there is already a value associated with `k`, that value is replaced.

`del d[k]` removes the key `k` from `d`.

`for k in d` iterates over the keys in `d`.

Lists and dictionaries are mutable!

Connecting the dots

Session 4

Outline

- Functions
- More complex programs
- Testing and debugging

Functions

Functions

- A **function** is a block of code which only runs when it is called.
- The idea is to encapsulate computation within a scope such that it can be treated as primitive:
  - Use by simply calling its name and providing input
  - Internal details are hidden from users
- Syntax:

```
def name of function(list of formal parameters):
    body of function
```

A simple example

- This code calculates the maximum of two numbers:

```
1  def maximum(x,y):
2      if x > y:
3          z = x
4      else:
5          z = y
6      return z
```

- We can then invoke by:

```
1  maximum(6,5)
```

  - When we call or invoke `maximum`(6,5), **`x`** is bound to 6, **`y`** is bound to 5, and then the body expressions are evaluated.

Function `return`

- The body of the function can consist of any number of expressions.
- Expressions are evaluated until:
  - Run out of expressions, in which case a special value `None` is returned
  - Or until special keyword `return` is reached, in which case subsequent expression is evaluated and that value is returned as value of function call

Summary of a function call

In [23]:

```
1  def maximum(x,y):
2      if x > y:
3          z = x
4      else:
5          z = y
6      return(z)
7
8  maximum(6,5)
```

Out[23]: 6

Function is defined

Function is invoked

1. The expressions for each parameter are evaluated, and bound to the formal parameters of the function.
2. The point of execution moves from the point of invocation to the first statement in the body of the function.
3. The code in the body of the function is executed until a `return` is encountered or there are no more statements.
4. The value of the invocation is the returned value.
5. The point of execution is transferred back to the code following the invocation.

Summary of function call

To recapitulate, when a function is called:

1. The expressions for each parameter are evaluated, and bound to the formal parameters of the function.
2. The point of execution moves from the point of invocation to the first statement in the body of the function.
3. The code in the body of the function is executed until a `return` is encountered or there are no more statements.
4. The value of the invocation is the returned value.
5. The point of execution is transferred back to the code following the invocation.

Scoping is local

- Each function defines a new space called a **scope**.
  - The formal parameters x, y and the local variable z that are used in `maximum` exist only within the scope of the function.

```
1  def maximum(x,y):
2      if x > y:
3          z = x
4      else:
5          z = y
6      return z
```

  - The assignments in `maximum` have no effect on the bindings of the names x, y, and z that exist outside the scope of the function.

Functions

jupyter

Activity 1: A function

Keyword arguments and default values

- Two ways that formal parameters get bound to actual parameters:
  - Positional arguments
  - Keyword arguments

```
1  def printName(firstName, lastName, reverse):
2      if reverse:
3          print(lastName + ', ' + firstName)
4      else:
5          print(firstName + ', ' + lastName)
```

```
1  printName('David', 'Smith', False)
2  printName('David', 'Smith', reverse=False)
3  printName('David', lastName='Smith',reverse=True)
4  printName(lastName='Smith', reverse=False, firstName='David')
```

- Keyword arguments are commonly used with default parameter values:

```
1  def printName(firstName, lastName, reverse=False):
2      if reverse:
3          print(lastName + ', ' + firstName)
4      else:
5          print(firstName + ', ' + lastName)
```

```
1  printName('David', 'Smith')
2  printName('David', 'Smith', True)
3  printName('David', 'Smith', reverse=True)
```

Specifications

- **Specifications** are a contract between implementer and user.
  - Assumptions: conditions that must be met by users of function (constraints on parameters)
  - Guarantees: what the function will provide, given that it has been called in a way that satisfies assumptions

```
1  def maximum(x,y):
2      """
3      Assumes x and y int or float
4      Returns the maximum of x and y
5      """
6      if x > y:
7          z = x
8      else:
9          z = y
10     return z
```

```
1  help(maximum)
```

```
Help on function maximum in module __main__:

maximum(x, y)
    Assumes x and y int or float
    Returns the maximum of x and y
```

Functions

Activity 2: Function with specifications

More complex programs

We have covered a lot!

- Object types

	SCALAR	STRUCTURED
IMMUTABLE	int float bool	string tuple
MUTABLE		list dict

- IF statements

```
if Boolean expression:
    {block of code}
elif Boolean expression:
    {block of code}
else:
    {block of code}
```

- Loops

```
while boolean expression:
    block of code
```

```
for variable in sequence:
    block of code
```

- Functions

```
def maximum(x,y):
    """
    Assumes x and y int or float
    Returns the maximum of x and y
    """
    if x > y:
        z = x
    else:
        z = y
    return z
```

Creating functions

jupyter

Activity 3

Activity 4

Activity 5

Activity 6

Activity 7

Testing and debugging

Syntax and static semantics errors

- To get your program to run, you must eliminate:
  - Syntax errors
    - English: "boy dog cat"
    - Python: "hi"5

```
1 "hi"5
```

```
  File "<ipython-input-7-569aaee41dee>", line 1
    "hi"5
       ^
SyntaxError: invalid syntax
```

  - Static semantic errors
    - English: "boy are cat"
    - Python: "hi"+5

```
1 "hi"+5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-723e511be247> in <module>
----> 1 "hi"+5

TypeError: must be str, not int
```

  - Valid statements
    - English: "boy hugs cat"
    - Python: "hi"+"5"

```
1 "hi"+"5"
```

```
'hi5'
```

- The Python interpreter will usually points these out: **Error**

Typical `error` messages

`IndexError`

Trying to access beyond the limits of a list/string/tuple

`TypeError`

Using data types in inappropriate ways

`NameError`

Referencing a non-existing variable

`SyntaxError`

Forgetting or misusing punctuation marks (parenthesis, colon, quotations…)

Semantics error

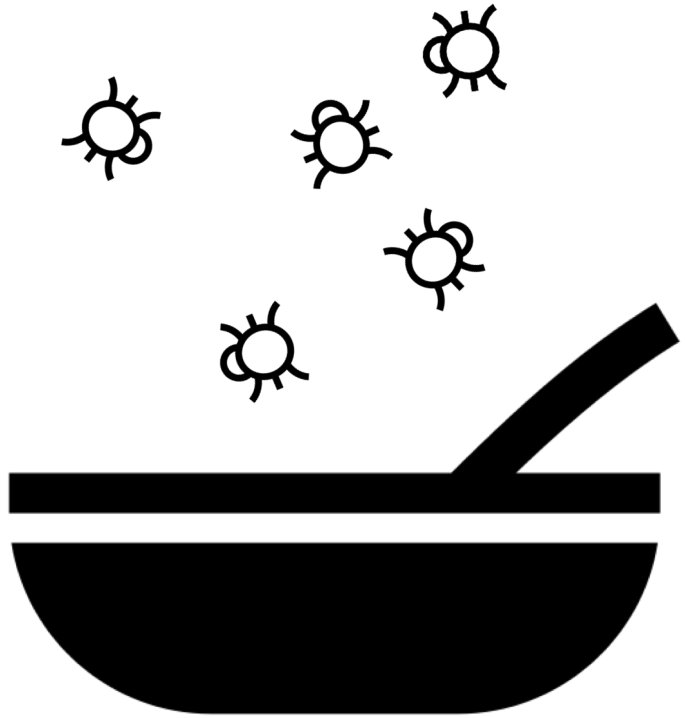
- **Semantics** is the meaning associated with a syntactically correct string of symbols with no static semantic errors
- If there is a semantic error in your program, it will run successfully (no error messages). However, your program will not do the right thing.

- The Python interpreter will **not** return an **`error`** in this case:
  - Program crashes
  - Program runs forever
  - Program gives an answer different than expected

Clean programs and clean soup, an analogy

You are making soup but bugs keep falling from the ceiling, what do you do?

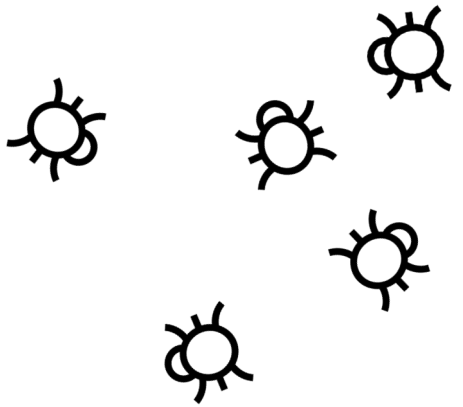
- Keep the lid closed
  - **Defensive programming**
- Check the soup for bugs
  - **Testing**
- Eliminate the source of bugs
  - **Debugging**



Analogy from "Computer science and programming using Python"

DEFENSIVE PROGRAMMING

- Write specifications for functions
- Modularize programs
- Document constraints and assumptions



TESTING

- Compare input/output pairs to specifications
- Does it work?
- How can I break my program?

DEBUGGING

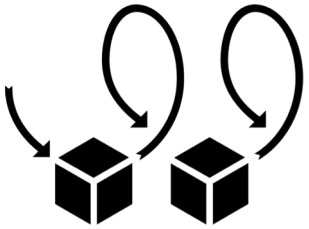
- Study events generating an error
- Why is it not working?
- How can I fix my program?

Defensive programming

- Good programmers design their programs in ways that make them easier to test and debug.

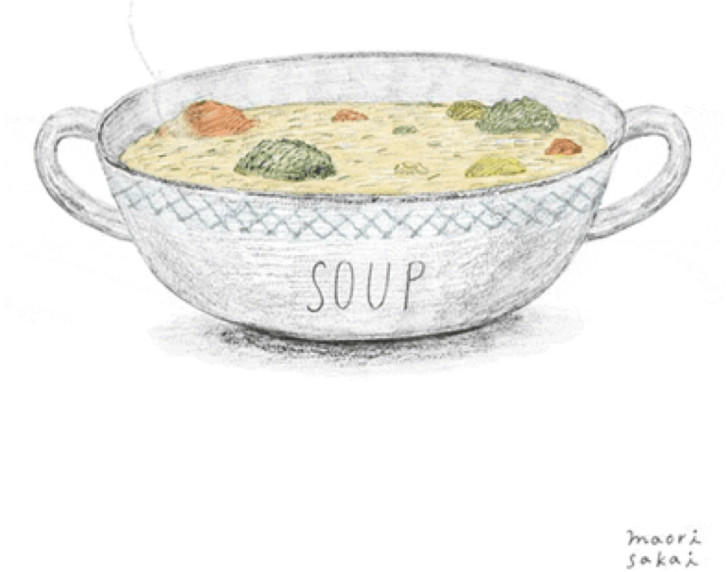
- The key is breaking the program up into separate components that can be implemented, tested and debugged independently.

- One way of modularizing programs is through functions.
  - Do not write, debug and test entire programs!
  - Do it function by function first, before the integration testing.



Testing

- **Testing** is the process of running a program to try and ascertain whether or not it works as intended.
- The purpose of a test is to show that a bugs exist.



"No amount of experimentation can ever prove me right;
a single experiment can prove me wrong."
A. Einstein

How to test?

- Come up with a set of expected results (input/output pairs)
  - But the simplest program has billions of possible inputs!
- Look for the natural partitions of the problem:
  - Partition the space of all possible inputs into subsets that provide equivalent information about the correctness of the problem

```
1  def isBigger(x, y):
2      """Assumes x and y are ints
3      Returns True if x is less than y and False otherwise."""
```

Sets of test values:	x positive, y positive	x = 0, y = 0
	x negative, y negative	x = 0, y ≠ 0
	x positive, y negative	x ≠ 0, y = 0
	x negative, y positive	

→ Reasonable probability of exposing a bug!

How to test?

- Black box testing: explore paths through specification
- Glass box testing: explore paths through code

Black box testing	Glass box testing
- Designed without looking at the code - Avoids biases - Can be reused if implementation changes - Guidelines: test cases for natural space partitions and boundary conditions	- Uses code to design test cases - Path-complete if every potential path through the code is tested at least once - Guidelines: conditionals (all branches), loops (not entered, executed once, executed several times), boundary conditions

Create test cases

jupyter

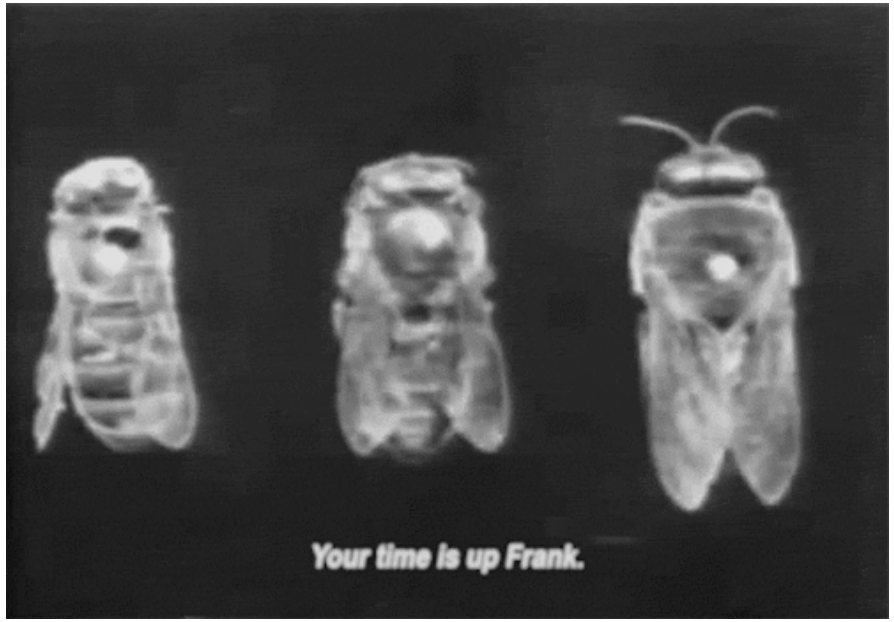
Activity 8: Testing

```
1  def absolut(x):
2      """
3      Assumes x is an int
4      Returns x if x>=0 and -x otherwise
5      """
6      if x< -1:
7          return (-x)
8      else:
9          return (x)
```

Debugging

- **Debugging** is the process of trying to fix a program that you already know does not work as intended.

- Now you must:
  - isolate the bug(s)
  - eradicate the bug(s)
  - retest until code runs correctly



Types of bugs

Bugs can be categorized according to two dimensions:

- **Overt | covert**
  - Overt is there is an obvious manifestation.
  - Covert if there is not an obvious manifestation.
- **Persistent | intermittent**
  - Persistent bugs occur every time the program is run with the same inputs.
  - Intermittent bugs occur only sometimes, seemingly under the same conditions.

Learning to debug

- **Be systematic** in your search:
  - Study available data
  - Form a hypothesis
  - Design and run an experiment to test your hypothesis
  - Keep record of the experiments performed

- Use `print` statements to test hypothesis
  - e.g. print function results
  - e.g. print variable values halfway into the code.

Debug this code

jupyter

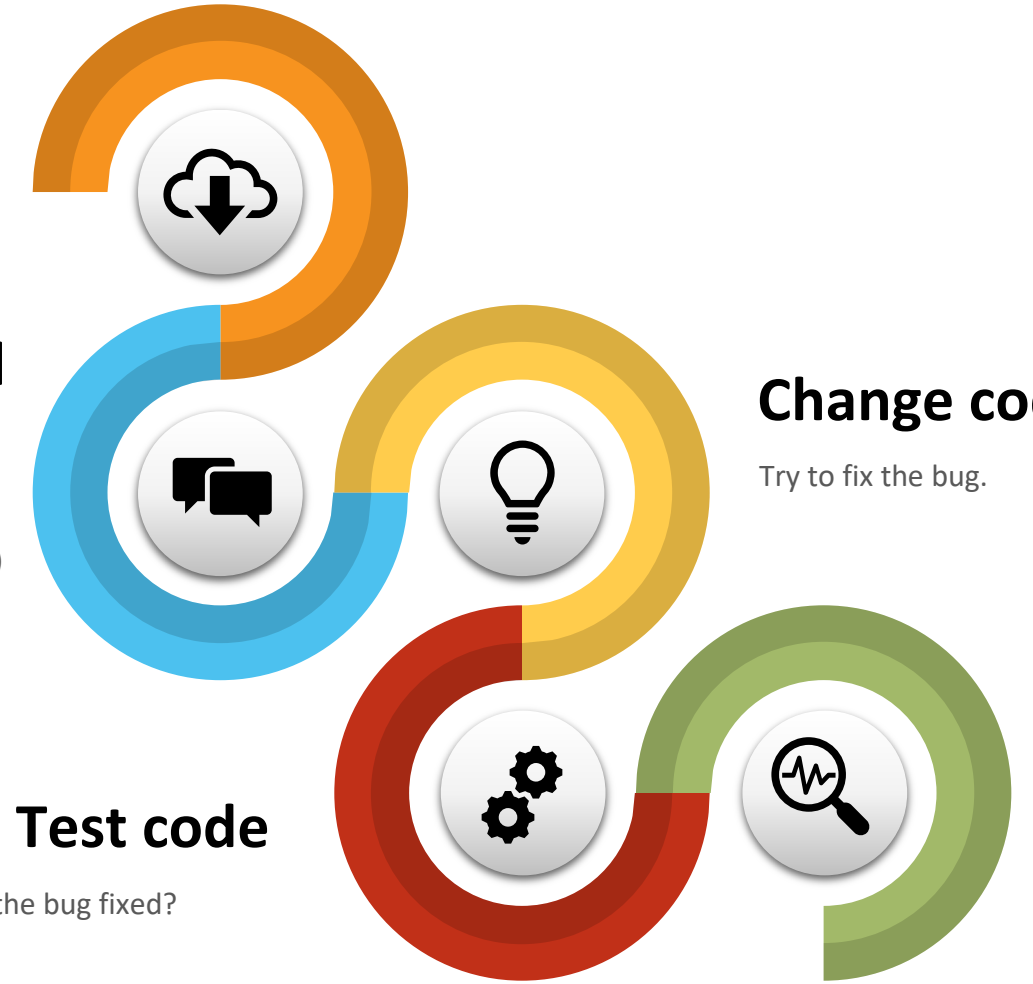
Activity 9: Debugging

```
1  def isPal(x):
2      """
3      Assumes x is a list
4      Returns True if the list is a palindrome, False otherwise.
5      A palindrome is a sequence that reads the same backward as forward.
6      """
7      temp = x
8      temp.reverse
9      if temp == x:
10         return True
11     else:
12         return False
13
14 def silly(n):
15     """
16     Assumes n is an int > 0
17     Gets n inputs from user
18     Prints "Yes" is the sequence of inputs forms a palindrome; "No"otherwise
19     """
20     for i in range(n):
21         result = []
22         elem = input("Enter element:")
23         result.append(elem)
24     if isPal(result):
25         print("Yes")
26     else:
27         print("No")
```

The debugging process

Backup code

DO NOT start editing the code before backing it up first!



Write down potential bug in a comment

Important to remember your hypothesis (What the bug was and where it was located)

Change code

Try to fix the bug.

Test code

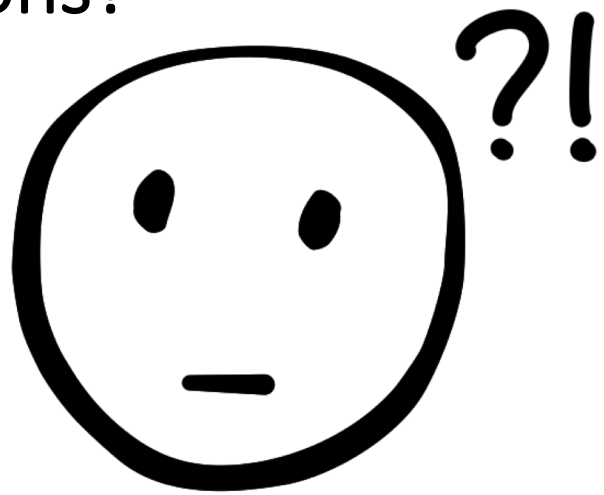
Is the bug fixed?

Compare new and old versions

Today we learned:

- Functions
- More complex programs
- Testing
- Debugging

Any questions?



References

Guttag, J. V. (2013). *Introduction to computation and programming using Python*. MIT Press.

Thanks!

Inma Borrella, Ph.D.
inma@mit.edu

PROBLEM 4.1

A sports shoe company in Oregon is going to establish a new line of product. The line will produce two types of products, one is advertised for strenght training and the other for running. Due to the difference in cushioning for each type of shoe and size, the company wants to know how many pairs of each type by size should produce. This will help them to plan their supply chain better and order the right amount of raw materials.

Sally is in charge of this new product line and she has received a dataset from another product line within the same company which has similar products. This dataset contains the historical demand of two types of shoes by size. Now Sally is asking you to perform the following exercises on the dataset.

Shoes Type	Size	Demand QTY
Running	14	3
Running	13.5	5

The dictionary Shoes (below) contains all the information from the table (shoes type, size and demand).

- Write a script that uses a for loop and an if clause to create two lists: list R for Running and list ST for Strenght Training. Each of these lists should contain all the information of the dictionary.
- In the list R, delete the string "Running" from all the elements of the list.
- Sort the elements of the list R in ascending order by size.

PROBLEM 4.2

Sam has recently joined a company as an analyst. The previous analyst, that he replaced, developed some data analysis code in python. However, when Sam is trying to run the code, some of the lines appear to fail. Can you help Sam and fix the lines below?

```
# This code is supposed to print the total cost S.
S = "$5350.50"
Print("The total cost is equal to ", S)
```

```
# This code is supposed to print numbers in ascending order.
for i in (1, 10):
print(i)
```

```
# This code is supposed to print result of applying the function cube to the variable x.

x = 2

def cube(Number):
    x = Number ** 3
    Number = Number + 1

print(x)
```

PROBLEM 4.3

Janette is having an on site interview with a global supply chain company. The firm expects the candidate to have a basic knowledge of python. One of the tasks during the onsite interview process is to be able to pass simple python questions.

Below is a sample of the test during the interview. Please help Janette to solve these problems.

Part 1: Using the lists Revenue and Cost provided in the cell below:

- Write a for loop to create the list Profit, that contains the daily profit for the last 7 days. Print the list Profit.
- Using a for loop and the list Profit, create a variable Weekly_Profit that contains the sum of all daily profits. Print the variable Weekly_Profit.

Part 2:

- Create a function that goes through a list and returns the index of the element with the highest value.
- Using the function you just created, in one line of code, print the maximum age from the list Age_of_Customers (provided in the cell below).

Part 3:

- Write a for loop for that goes through the list Age_of_Customers and prints the first element that is smaller than 19 along with its index. Also stop the loop immediately after the print.
- Using a while loop, count the number of customers older than 50 and print the final number.