CSCI 134 Fall 2021:
Functions

Sept 15, 2021

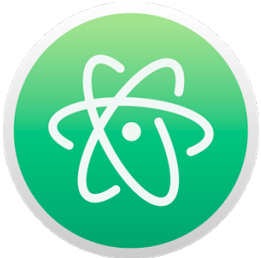
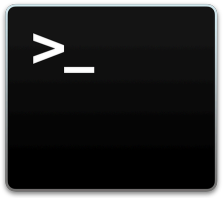
Shikha Singh, 9AM
Jeannie Albrecht, 10AM

Check-in After First Lab!

- You have all survived your first computer science lab
  - **Congratulations!**
- Computer science tools that you used:
  - **Atom** as a text editor
  - **Terminal** as a text-based interface to the computer
  - **Git** for versioning, **Github/Gitlab** (cloud-based hosting service) for retrieving & submitting your work
  - **Python**, of course

Do You Have Any Questions?



Announcements & Logistics

- **Lab 1** due today at 10 pm (for Monday labs)
- **Lab 1** due tomorrow at 10 pm (for Tuesday labs)
- **Homework 2** released today, due next Monday at 10 pm
- **Office hours and TA hours today**
  - Shikha: 12:30 - 2:30pm
  - Jeannie: ~~1-3 pm~~ 2-4pm
  - Lida: 2-4 pm
  - TAs 4-6 pm and 7-11 pm in TCL 217A and TCL 216
- **Herd scheduling**: We got your info, we are working on it!

Do You Have Any Questions?

Aside: Accessing Lecture Materials

CSCI 134 - Fall 2021

Introduction to Computer Science

Home | Lectures | Labs & HW | Resources | CS@Williams

Lectures

Readings. The lectures will initially be supported by the text Think Python (TP) 2e. As the course progresses, the lecture slides and jupyter notebooks (posted below) will take the place of the textbook.

Jupyter Notebooks. We will be using Jupyter notebooks in this course. Jupyter notebooks allow us to have a rich web-based interface to run interactive python examples. The notebook for each lecture will be distributed here in the form of a zip file containing the html file, and finally the source 'ipynb' (read as interactive python notebook) file.

How to read Jupyter Notebooks. Typing a command in a 'In[]' cell in a Jupyter notebook is the same as typing it in an interactive python session. The 'Out[]' cell of the notebook gives the resulting output. Thus, a Jupyter notebook is essentially an enhanced way to use interactive python: it stores code examples that can be executed live and is rendered in a rich format.

Installing Jupyter Notebooks. You may view the static notebook from class as an html file, but if you would like to run the examples dynamically you should download and install jupyter notebooks by following the instructions here.

Date	Topic & Reading	Shikha's (9 am) Lecture	Jeannie's (10 am) Lecture	Jupyter Notebook
Sept 10	Welcome & Logistics (TP Ch 1)	Slides	Slides	N/A
Sept 13	Types & Expressions (TP Ch 2)	Slides	Slides	[html], [zip]

Last Time

- Discussed **data types** and **variables** in Python
  - int, float, boolean, string
- Learned about basic operators
  - arithmetic, assignment
- Experimented with built-in Python functions
  - int(), input(), print()
- Investigated different ways to run and interact with Python

Review: Jupyter Notebooks

- How can you experiment with examples that we do in class with a **Jupyter notebook** by yourself?
    - We recommend running the examples at the end of Lecture 2 that we didn't cover in class
    - Reviewing these notebooks is a great way to review lecture material

Today

- We will discuss functions in greater detail
- Review the built-in functions we (briefly) saw last time and in lab
  - `input()`, `print()`, `int()` all expect argument(s) within the parens
  - We will examine these a bit more today
- We will discuss the distinction between fruitful and non-fruitful functions
- We will also learn how to define our own functions

Review:
Python Built-in Functions

`int(), float(), str()`

`input(), print()`

Built-in functions: int()

- When given a string that's a sequence of digits, optionally preceded by `+/-`, `int()` returns the corresponding integer
- On any other string it raises a `ValueError`
- When given a float, `int()` returns the integer that results after **truncating** it towards zero
- When given an integer, `int()` returns that same integer

```
In[1] int('42')
Out[1] 42
In[2] int(-5.5)
Out[2] -5
In[3] int('3.141')
ValueError
```

Built-in functions: float()

- When given a string that's a sequence of digits, optionally preceded by `+/-`, and optionally including one decimal point, `float()` returns the corresponding floating point number.
- On any other string it raises a `ValueError`
- When given an integer, `float()` converts it to a floating point number.
- When given a floating point number, float returns that number

```
In[1] float('3.141')
Out[1] 3.141
In[2] float('-273.15')
Out[2] -273.15
In[3] float('3.1.4')
ValueError
```

Built-in functions: str()

- Converts a given type to a `string` and returns it
- Returns a syntax error when given invalid input

```
In[1] str(3.141)
Out[1] '3.141'
In[2] str(None)
Out[2] 'None'
In[3] str(134)
Out[3] '134'
In[4] str($)
SyntaxError: invalid syntax
```

Built-in functions: input()

- `input()` displays its single argument as a prompt on the screen and waits for the user to input text, followed by **Enter/Return**
- It returns the entered value as a **string**

```
In[1] input('Enter your name: ')
Enter your name: Harry Potter
Out[1] 'Harry Potter'
In[2] age = input('Enter your age : ')
Enter your age: 17
In[3] age
Out[3] '17'
```

Prompts in red. User input in blue.
Inputted values are by default a **string**

Built-in functions: print()

- `print()` displays a character-based representation of its argument(s) on the screen and returns a special **`None`** value (not displayed). Notice there are no "Out[]" lines.

```
In[1] name = 'Harry Potter'
In[2] print('Your name is', name)
Your name is Harry Potter
In[3] age = input('Enter your age : ')
Enter your age: 17
In[4] print('The age of ' + name + ' is ' + age)
The age of Harry Potter is 17
```

Comma as a separator adds a space

Can also add spaces through string *concatenation*

Today:
User-Defined Functions

Structuring Code

- So far we have:
  - Written simple expressions
  - Created small scripts to perform certain tasks
- This is fine for small computations!
  - But we need more organization for larger problems
- Structuring code is good for:
  - Keeping track of which part of our code is doing what actions
  - Keeping track of what information needs to supplied where
  - **Reusability!** Specifically, reusing blocks of code

Abstracting with Functions

- **Abstraction**: Reduce code complexity by ignoring (or hiding) some implementations details
  - Allows us to achieve code decomposition and reuse
- Real life example: a projector
  - We know how to switch it on and off **(public interface)**
  - We know how to connect it to our computer **(input/output)**
  - We don't know how it works internally **(information hiding)**
- **Key idea:** We don't need to know much about the internals of a projector to be able to use it!
  - Same is true with functions!

Decomposition

- To write organized code, divide individual tasks into separate functions
  - Functions are **self-contained**
  - Each function is a **small piece** of a **larger task**
  - Functions are **reusable**
  - Keep code **organized**
  - Keep code **coherent**
- We have already seen some built-in examples (int(), input(), print(), etc)
- Today we will learn how to decompose our Python code and hide small details using *user-defined functions*
- Later in the semester, we will learn a new abstraction which achieves a greater level of decomposition and code hiding: classes

Anatomy of a Function

- Function **definition** characteristics:
  - Has a **name** #header
  - Has **parameters** (optional) #header
  - Has a **docstring** (optional, but recommended) #header
  - Has a **body** (indented and required)
  - Always **returns** something (with or without an explicit `return` statement)
- Statements within the body of a function are not run in a program until they are "called" or "invoked" through a **function call** (like calling print() or int() in your program)

Function Example

Function definition

Function's name is `square`

```
def square(x):
    '''Takes a number and returns its square'''
    return x*x
```

Function Calls/Invocations

```
In [1] square(5)
Out [1] 25
In [2] square(-2)
Out [2] 4
```

Function Example

`square` has one **parameter**, `x`, which is the expected input to the function.

Function definition

```
def square(x):
    '''Takes a number and returns its square'''
    return x*x
```

Function Calls/Invocations

```
In [1] square(5)
Out [1] 25
In [2] square(-2)
Out [2] 4
```

Function Example

Function definition

This is the **docstring**, which is enclosed in triple quotes. It is a short description of the function.

```
def square(x):
    '''Takes a number and returns its square'''
    return x*x
```

Function Calls/Invocations

```
In [1] square(5)
Out [1] 25
In [2] square(-2)
Out [2] 4
```

Function Example

Function definition

```
def square(x):
    '''Takes a number and returns its square'''
    return x*x
```

This is the body of the function. Notice that this functions includes an explicit **return** statement.

Function Calls/Invocations

```
In [1] square(5)
Out [1] 25
In [2] square(-2)
Out [2] 4
```

Function Example

Function definition

```
def square(x):
    '''Takes a number and returns its square'''
    return x*x
```

Notice the indentation. This is very important!!

Function Calls/Invocations

```
In [1] square(5)
Out [1] 25
In [2] square(-2)
Out [2] 4
```

Function Example

Function definition

```
def square(x):
    '''Takes a number and returns its square'''
    return x*x
```

When we call/invoke the function, 5 is the **argument** value.

Function Calls/Invocations

```
In [1] square(5)
Out [1] 25
In [2] square(-2)
Out [2] 4
```

Function Example

Function definition

```
def square(x):
    '''Takes a number and returns its square'''
    return x*x
```

Function Calls/Invocations

```
In [1] square(5)
Out [1] 25
In [2] square(-2)
Out [2] 4
```

Summary:

- Indent in function body (required)
- Colon after function name (required)
- Docstring (recommended, good style)
- `x` in function definition is a parameter
- Single line body which returns the result of the expression `x * x`
- `return` always ends execution!
- Function is defined once and can be called any number of times!

A Closer Look At Parameters

- **Parameters** are "holes" in the body of a function that will be filled in with **argument values** in each invocation
- A particular name for a parameter is irrelevant, as long as we use it consistently in the body (just like f(x) and f(y) in math)
  - All of the square function definitions work exactly the same way!
  - Invocation would also look exactly the same: `square(5)`

```
def square(x):
    return x*x
```

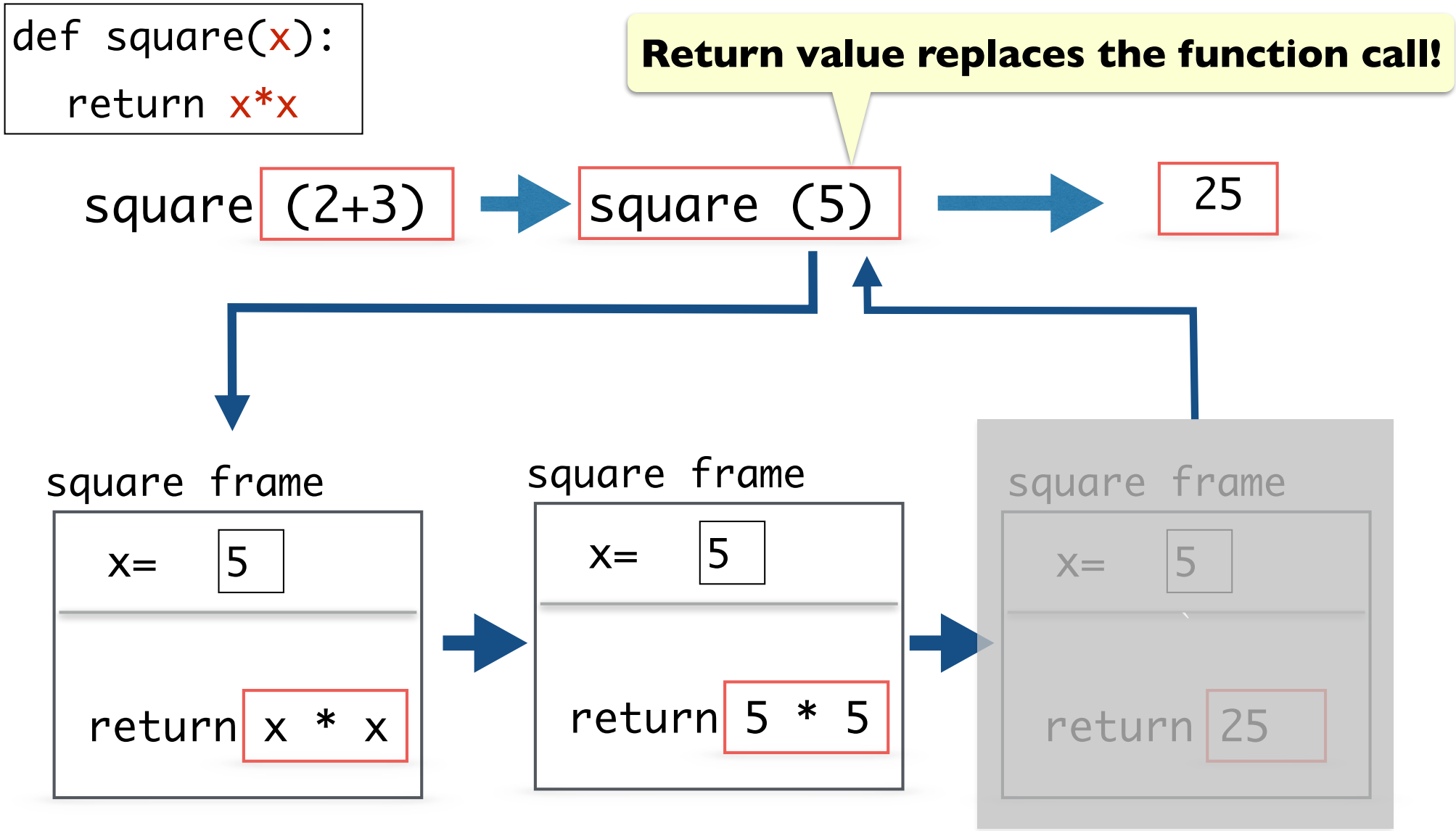
```
def square(apple):
    return apple*apple
```

```
def square(num):
    return num*num
```

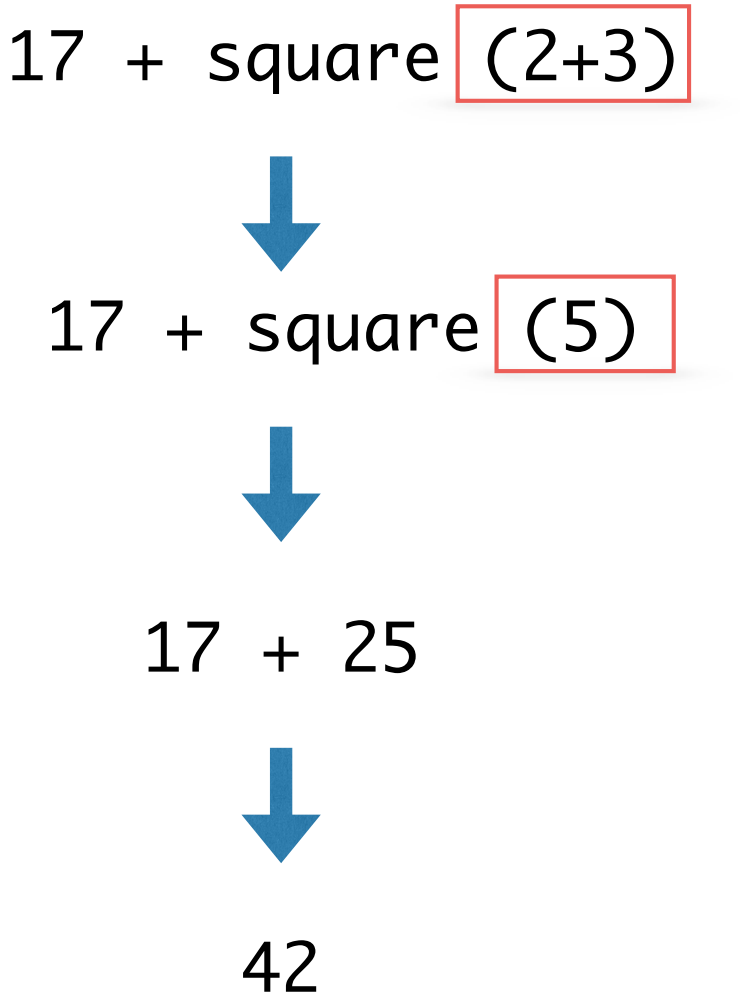
Rule of thumb: Choose parameter names that make sense. Avoid always using x, for example.

Python Function Call Model

Function frame: Model for understanding how a function call works



Function Call Replaced by Return Value



Jupyter Notebook:
Let's See Some Examples