Functions

COMP1730/6730

Reading: Textbook chapter 3 : Alex Downey, *Think Python*, 2nd Edition (2016)

Or: Chapter 3: Al Sweigart, *Automate the boring stuff with python*, 2nd Edition (2020)

Australian
National
University

Functions (*Think Python*, Ch. 3)

Australian National University

- Functions are like mini-programs you can call from your code that do useful, predefined tasks (that you would otherwise you might need to do for yourself)
- We have already seen two:
  - `str(0.1)` converts a number (integer or float) to a string
  - `type(153)` prints the variable type
  - `print('this')` takes a string input and prints it to the terminal
- In python, functions can be:
  - Built-in
  - Imported from modules
  - User-defined

Built-in Python functions

Australian National University

- There LOTS of these. A very necessary part of the language
- Go to the source documentation for the Python language (at python.org) and look through what is available:

https://docs.python.org/3/library/functions.html

- Each function is described with details of what it does, what input it takes and what output it produces

Built–in Functions

A	E	L	R
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
any()			round()
anext()	F	M	
ascii()	filter()	map()	S
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	G	N	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	H	object()	
callable()	hasattr()	oct()	T
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		P	V
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	Z
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		_
	iter()		__import__()

Example Built-in functions:

`print(), len(), round(), input()`

Australian National University

- As a useful exercise, go to the python.org documentation for built-in functions and look up these up

```
>>>
>>> print('Some text here')
Some text here
>>> len('Some text here')
14
>>> round(1.1)
1
>>> round(1.9)
2
>>> input_string = input()
here is something I typed
>>> print(input_string)
here is something I typed
>>>
```

Imported Functions

Australian National University

- These work a lot like built-in functions, but need to be imported first and called with reference to the module they come from.
- Sometimes modules are referred to as packages or libraries
- A full list of Python modules is available at python.org:

https://docs.python.org/3/py-modindex.html

- Have a look at what is available – lots of useful stuff, eg:

math	json	pickle
random	statistics	getopt
zlib	pprint	csv

Python Module Index

_ | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | z

_	
__future__	*Future statement definitions*
__main__	*The environment where top-level code is run. Covers command-line interfaces, import-time behavior, and ``__name__ == '__main__'``.*
_thread	*Low-level threading API.*
a	
abc	*Abstract base classes according to :pep:`3119`.*
aifc	**Deprecated:** *Read and write audio files in AIFF or AIFC format.*
argparse	*Command-line option and argument parsing library.*
array	*Space efficient arrays of uniformly typed numeric values.*
ast	*Abstract Syntax Tree classes and manipulation.*
asynchat	**Deprecated:** *Support for asynchronous command/response protocols.*
asyncio	*Asynchronous I/O.*
asyncore	**Deprecated:** *A base class for developing asynchronous socket handling services.*
atexit	*Register and execute cleanup functions.*
audioop	**Deprecated:** *Manipulate raw audio data.*
b	
base64	*RFC 4648: Base16, Base32, Base64 Data Encodings; Base85 and Ascii85*
bdb	*Debugger framework.*
binascii	*Tools for converting between binary and various ASCII-encoded binary representations.*

Module functions example: math

Australian National University

- The functions available in modules would be rather annoying to have to write from scratch each time: `math.log10()`, `math.sin()`, etc
- To use a particular module, one must first import it:

```
>>> import math
```

- Then to use the functions, one must use dot notation:

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

Downey (2015) *Think Python*, 2nd Ed.

How to find out more

Australian National University

`docs.python.org` are your friend

This often the best way to find out how things work – the docs written by the developers.

If we scroll down this page far enough, the `math.sin()` is mentioned, as well as other useful things.

https://docs.python.org/3/library/math.html

`math.hypot(*coordinates)`
Return the Euclidean norm, `sqrt(sum(x**2 for x in coordinates))`. This is the length of the vector from the origin to the point given by the coordinates.

For a two dimensional point `(x, y)`, this is equivalent to computing the hypotenuse of a right triangle using the Pythagorean theorem, `sqrt(x*x + y*y)`.

Changed in version 3.8: Added support for n-dimensional points. Formerly, only the two dimensional case was supported.

Changed in version 3.10: Improved the algorithm's accuracy so that the maximum error is under 1 ulp (unit in the last place). More typically, the result is almost always correctly rounded to within 1/2 ulp.

`math.sin(x)`
Return the sine of *x* radians.

`math.tan(x)`
Return the tangent of *x* radians.

Angular conversion

`math.degrees(x)`
Convert angle *x* from radians to degrees.

`math.radians(x)`
Convert angle *x* from degrees to radians.

Hyperbolic functions

Hyperbolic functions are analogs of trigonometric functions that are based on hyperbolas instead of circles.

`math.acosh(x)`
Return the inverse hyperbolic cosine of *x*.

Exercises

Australian National University

- Complete Exercises 3-1, 3-2 and 3-3 of *Think Python*.

Reading

- Chapter 3 of *Think Python* AND/OR
- Chapter 3 of *Automate the Boring Stuff with Python*

Writing your own functions

Australian National University

- Why? If there are parts of your code that you use over and over in a single program, it makes good sense to turn these into a helper function:
  - Shorter code
  - Your code will be easier to read and understand
  - And – the best reason – you only need to change your code in one place when you modify it! It is annoying and very bug-prone to have to make the same changes is multiple different places in your code
  - Eventually, you will end up with a group of helper functions specific to your own work. And you will end up using these over-and-over.

A simple, custom function

Australian National University

- Function definitions start with:
  - the `def` keyword
  - have a name followed by parentheses `()`
  - and a colon `:`
- First this is the definition line. It is followed by an indented code block that does the work of the function:

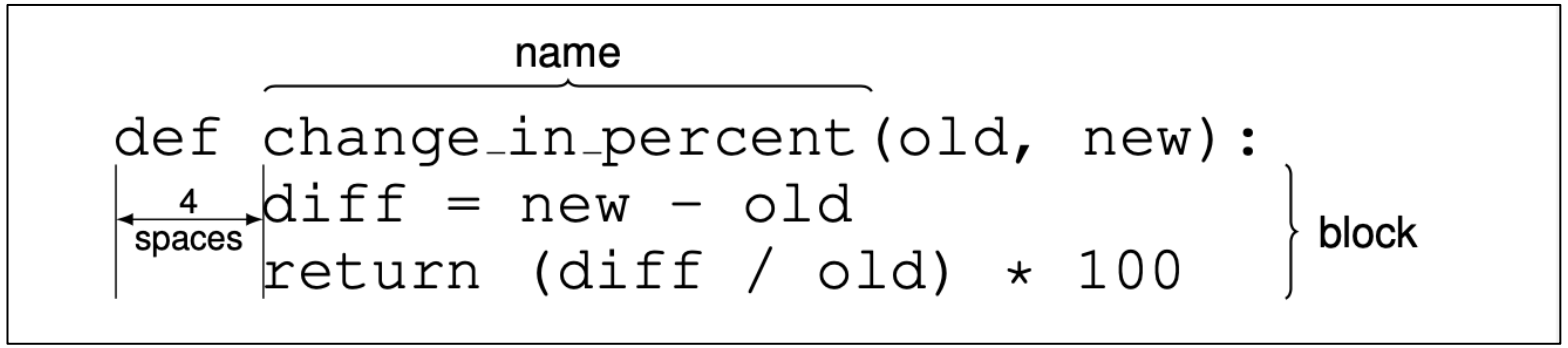
```
>>> def make_a_sound():
...     print('quack')
...
>>> make_a_sound()
quack
```

Lubanovic (2019) *Introducing Python*

- Functions are called by their name, with parentheses `()`
- A function must always be defined before it is called

Function definition

Australian National University



- A function definition consists of a name and a body (a block)
- The extent of the block is defined by indentation, which must be the same for all statements of a block
  - Standard indentation is 4 spaces
- This example has **parameters**
  - Parameters are specified in the function call, and are passed to the code block
- A custom function must be defined before it can be called

Function parameters and `return` value

Australian National University

parameters

```
def change_in_percent(old, new):
    diff = new - old
    return (diff / old) * 100
```

- Function (formal) parameters are (variable) names
  - These variables can be used only in the function body
- Parameter values will be set only when the function is called
- `return` is a statement
  - when executed, it causes the function call to end, and return the value of the expression

A function call – with parameters

Australian National University

- To call a function, write its name followed by its (actual) **arguments** in parentheses:

```
>>> change_in_percent(489, 556)
13.701431492842536
```

- The arguments are expressions
- Their number should match the parameters
  - Though there can be exceptions – more about this later
- A function call is an expression
  - The call in the example above is an expression that evaluates to the value `return`'d by the function

Terminology: arguments and parameters

Australian National University

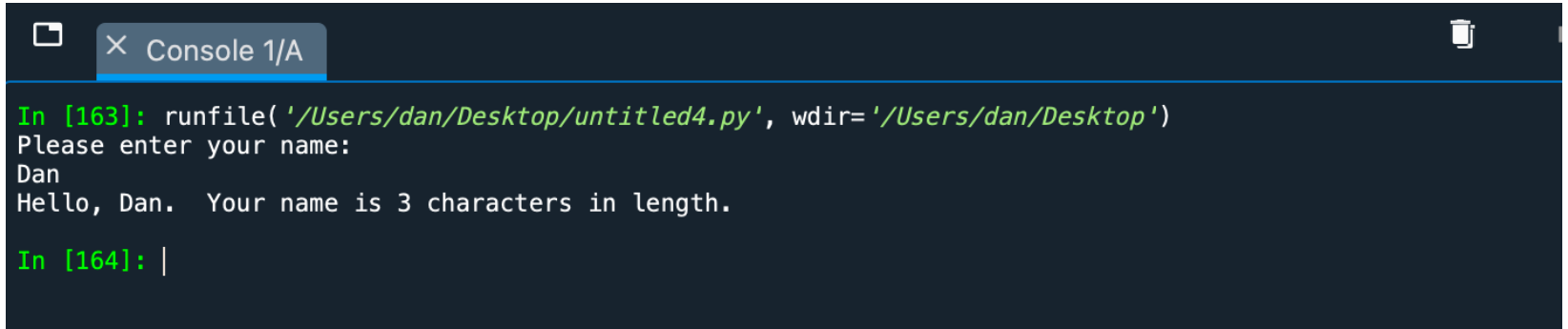
- Arguments are values that are passed to a function when it is called
- Say we make this function call:
  - `print("ATGTAATAG")`
  - `print()` is the function
  - `"ATGTAATAG"` is the string argument passed to `print()`
- Parameters are what arguments become when inside the code block within the function

Functions can call other functions

Australian National University

- This is what real-world code is doing all the time. Most code you will write will use other functions to get things done

```
def ask_name():
    print("Please enter your name: ")
    name = input()
    return name

def calculate_length_of_string(the_string):
    return len(the_string)

def print_greeting(input_name):
    name_length = calculate_length_of_string(input_name)
    print("Hello, " + input_name + ".  Your name is " + str(name_length) + " characters in length.")

def interact():
    interaction_name = ask_name()
    print_greeting(interaction_name)

interact()
```



Function definition order

Australian National University

- A function must be defined before it is first called.
- Not like:

```
interact()

def ask_name():
    print("Please enter your name: ")
    name = input()
    return name

def calculate_length_of_string(the_string):
    return len(the_string)

def print_greeting(input_name):
    name_length = calculate_length_of_string(input_name)
    print("Hello, " + input_name + ".  Your name is " + str(name_length) + " characters in length.")

def interact():
    interaction_name = ask_name()
    print_greeting(interaction_name)
```

Moved function call to program beginning

```
Console 1/A

In [164]: runfile('/Users/dan/Desktop/untitled4.py', wdir='/Users/dan/Desktop')
Traceback (most recent call last):

  File "/Users/dan/opt/anaconda3/lib/python3.9/site-packages/spyder_kernels/py3compat.py", line 356, in co
    exec(code, globals, locals)

  File "/Users/dan/Desktop/untitled4.py", line 1, in <module>
    interact()

NameError: name 'interact' is not defined
```

Order of evaluation

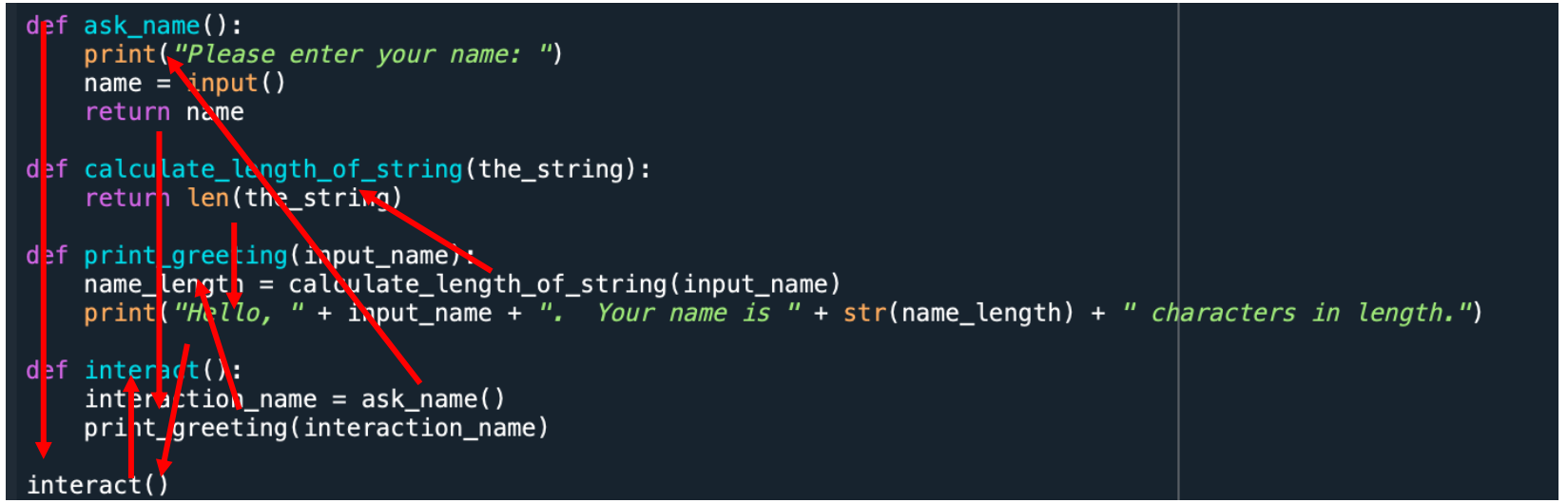
Australian National University

- The python interpreter always executes instructions one at a time in sequence; this includes expression evaluation
- To evaluate a function call, the interpreter:
  - First, evaluates the argument expressions one at a time, from left to right
  - Then, executes the function body with its parameters assigned the values returned by the arguments expressions
- Same with operators: first arguments (left to right), then the operation

Flow of execution

Australian National University

- Calling a function will interrupt the processive flow of program execution
- Calling a function causes the execution to skip to that function and continue executing from that position



- Execution continues until the end of the function is reached (and it returns to executing where the call was originally made)

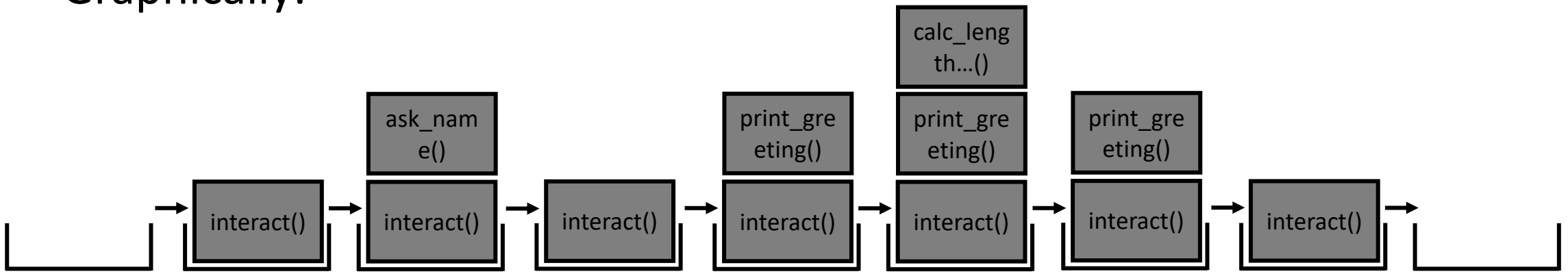
Concept: the call stack

Australian National University

- The ‘to-do list’ of where to come back to after each current function call is called the (execution or call) stack
- When evaluation of a function call begins, the current instruction sequence is put ‘on hold’ while the expression is evaluated – and the function calls begin to ‘stack up’

```
def ask_name():
    print("Please enter your name: ")
    name = input()
    return name

def calculate_length_of_string(the_string):
    return len(the_string)

def print_greeting(input_name):
    name_length = calculate_length_of_string(input_name)
    print("Hello, " + input_name + ".  Your name is " + str(name_length) + " characters in length.")

def interact():
    interaction_name = ask_name()
    print_greeting(interaction_name)

interact()
```

- Graphically:



Scope - Sweigart, *Automate the Boring Stuff with Python,* Ch 3

Australian National University

- We haven't talked yet about scope – this is important
- So far, we have assumed that all defined variables are accessible all the time – this is known as **global** scope
- But global scope becomes hazardous as:
  - A program gets larger
  - Includes code that comes from other developers (you might both use the same variable name)
- The parameter variables within a single function are **local** to the code block. If you try to access one of these outside the function code block, you will get an error.

Graphically:

Australian National University

scope.py

```
def print_gene():
    gene_name = 'p53'
    print('In print_gene: ' + gene_name)

def print_protein():
    protein_id = 'TP53'
    gene_name = 'Unknown'
    print('In print_protein: ' + protein_id + ' ' + gene_name)


gene_name = 'BRCA2'
print_gene()
print('In main: ' + gene_name)
print_protein()
```

Output:

```
In print_gene: p53
In main: BRCA2
In print_protein: TP53 Unknown
```

Program: scope.py

Global

`gene_name = 'BRCA2'`

Function: print_gene()

Local

`gene_name = 'p53'`

Function: print_protein()

Local

`protein_id = 'TP53'`
`gene_name = 'Unknown'`

Within a function, parameters are local

Australian National University

- Variables created/assigned in a function (including parameters) are **local** to that function:
  - Local variables have scope limited to the enclosing block
  - The interpreter uses a new namespace for each function call
  - Local variables that are not parameters are undefined before the first assignment in the function body. Then remain local to the function block
  - Variables with the same name used outside the function are unchanged after the call
- Within a function, you can still access variables in the **global** scope
- But, within function **local** scope, you *cannot* access the **local** scopes of other functions

Why make it complicated?

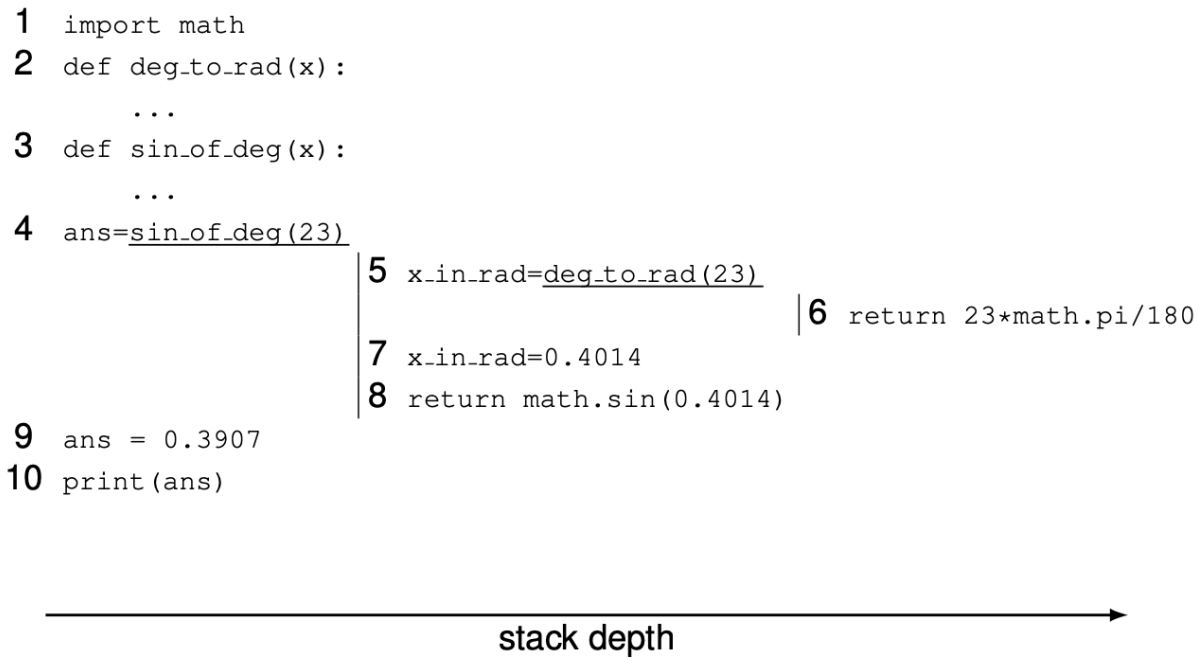
Australian National University

- There are very good reasons why every section of code should not be able to access the variables controlled by other sections.
  - For one thing, as your program gets bigger, the **namespace** of the program will start to get crowded.
  - You might be using the same variable name for two different things.
  - If you are using code from other developers (like importing functions), they might be using the same variable names as your program – but for different things
  - It makes good sense to compartmentalise variable scope, to avoid *namespace-collisions*

The call stack

Australian National University

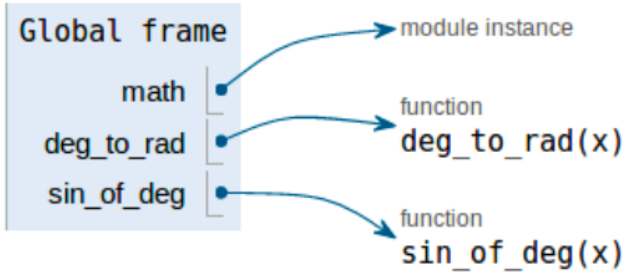
```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```



The call stack and scope

Australian National University

```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```

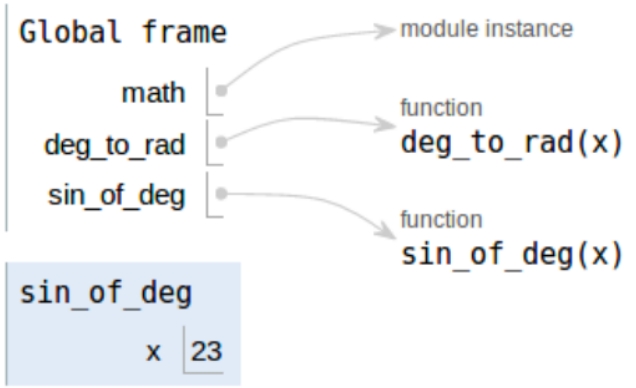


(Image from pythontutor.com)

The call stack and scope

Australian National University

```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```

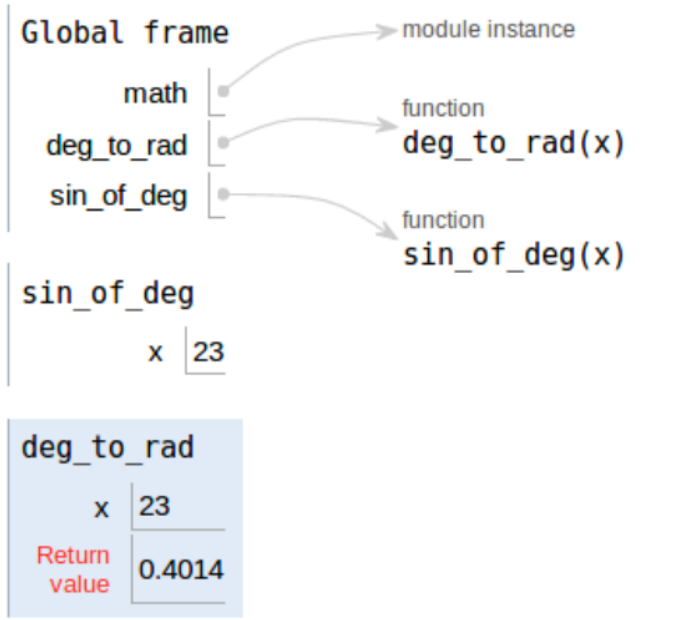


(Image from pythontutor.com)

The call stack and scope

Australian National University

```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```

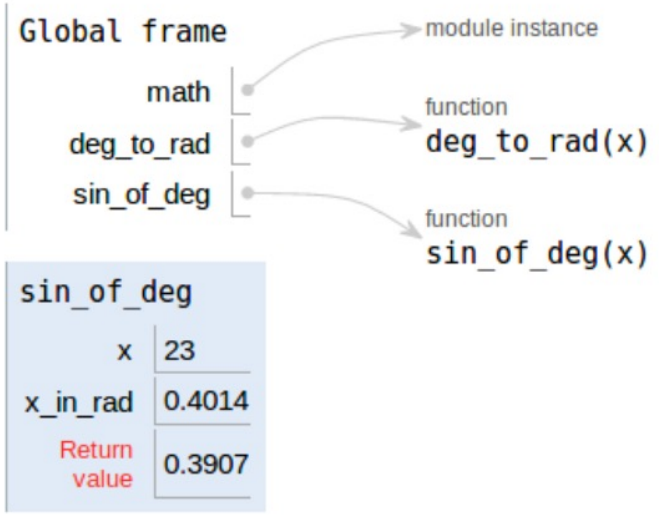


(Image from pythontutor.com)

The call stack and scope

Australian National University

```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```

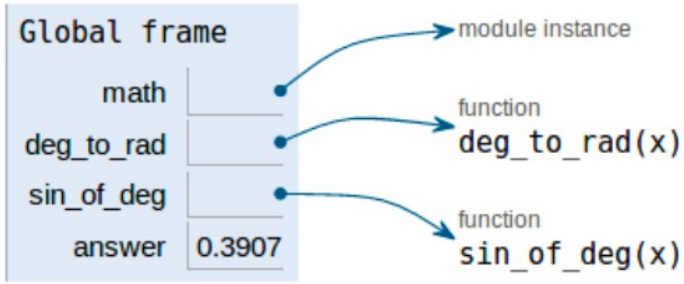


(Image from pythontutor.com)

The call stack and scope

Australian National University

```
import math
# Convert degrees to radians
def deg_to_rad(x):
    return x * math.pi / 180

# Take sin of an angle in degrees
def sin_of_deg(x):
    x_in_rad = deg_to_rad(x)
    return math.sin(x_in_rad)

ans = sin_of_deg(23)
print(ans)
```



(Image from pythontutor.com)

`None` values

Australian National University

- Some variables contain nothing. Not zero. `None` means null, nothing, undefined.
- `None` type:

```
>>> print(type(None))
<class 'NoneType'>
```

Downey (2015) Think Python, 2nd Ed., Ch 3

- Not the same as zero:

```
>>>
>>> none_var = None
>>>
>>> none_var == 0
False
>>> _
```

- A void value. Just not defined. Some other languages have `NULL` values.
- Why are `NoneType` values useful?

Functions withOUT `return` values

Australian National University

- One place you might encounter `None` is when a function has no `return` statement
- If execution of a function reaches the end of the body without encountering a return statement, the function call returns `None`

- *Note*: with iPython, or interactive mode with Spyder, the interpreter does not print the return value of an expression when the value is `None`.

The function docstring

Australian National University

- It is good practice to document your function with a *docstring*
- As simple as a sentence bound with `'''`

```
def change_in_percent(old, new):
    '''Return change from old to new, as a percentage of the old value.
    Old value must be non-zero.'''
    return ((new - old) / old) * 100
```

- A docstring is a string literal written as the first statement inside a function's body
- Acts like a comment, but accessible through the built-in help system
- Describe *what* the function does (if not obvious from its name) – and its *limits* and *assumptions*

Exercises

Australian National University

- Complete Exercises 3-1, 3-2 and 3-3 of *Think Python*.
- And the Practice Project ‘The Collatz Sequence’ in *Automate the Boring Stuff with Python*, at the end of Chapter 3

Reading

- Chapter 3 of *Think Python* AND/OR
- Chapter 3 of *Automate the Boring Stuff with Python*