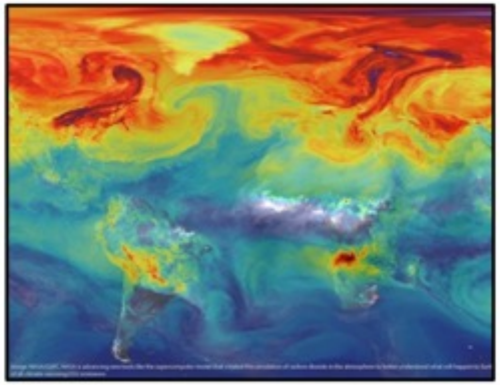


Geo Data Science with Python (GEOS-5984/4984)

Prof. Susanna Werth

Topic: Python Statements - Functions

Today's music is from: Carmen

Please keep sending me your song suggestions through Canvas!

Notes/Reminders

- Reading notebooks – purpose: for revising any content you did not get the first time or if you are curious for more details

- Revise Indefinite loops and loop breakers …

General `while` Statements

```
while test:
    statements
    break
    continue
    pass
else:
    statements
```

- `else`
- Compound statements
- Loop breakers

Loop Breakers

- `Break`
  - Terminates loop
  - For and while loops
- `Continue`
  - Returns back to start of loop statement
  - For and while loops
- `Pass`
  - Used when syntactically a statement is required, but don’t want to anything to happen

Examples

- Break
- Continue
- Pass

Tutorial

Iteration 2 & 3: Check the password!

1. Define a password string.
2. Create a `while` loop, which asks for the user to input a password using the function ‘input()’.
3. While going through this loop, there are two possible outcomes:
   - If the password is correct, the while loop will exit.
   - If the password is not correct, the while loop will continue to execute.
4. + ELSE: Add final feedback to the user by coding an else statement for the while loop, when it is exited.
5. + COMOUND STATEMENTS (optional):
   - Add a counter and interrupt the password input after three false trials.
   - Return final feedback whether password is correct or not.
6. + LOOP BREAKERS

Indefinite Loops

Note:

*This is only possible straightforwardly with *indefinite* loops.*

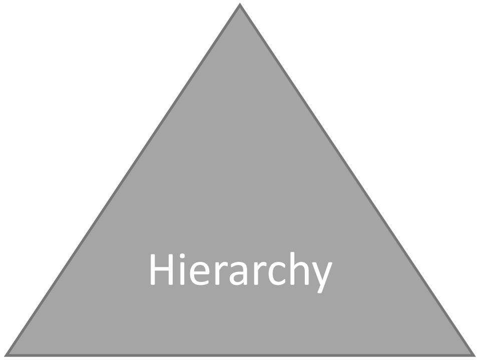
Don't mix this up with infinite loops.

Today

- Functions
- Function Scopes and Namespaces

Python Conceptual Hierarchy

- Python program components
  - Programs are composed of ***modules***
  - Modules contain ***statements***
  - Statements contain ***expressions***
  - Expressions create and process ***objects***



- *Objects* are data elements (e.g. variables, functions, ...)
- *Expression* is a **combination of one or more objects** that the programming language interprets and computes to **produce another object**. They are embedded in statements.
- *Statements* code the larger logic of a program (e.g. assignment, selections, iteration...)
- *Modules* are highest-level organization unit, packages code for reuse

Python Statements

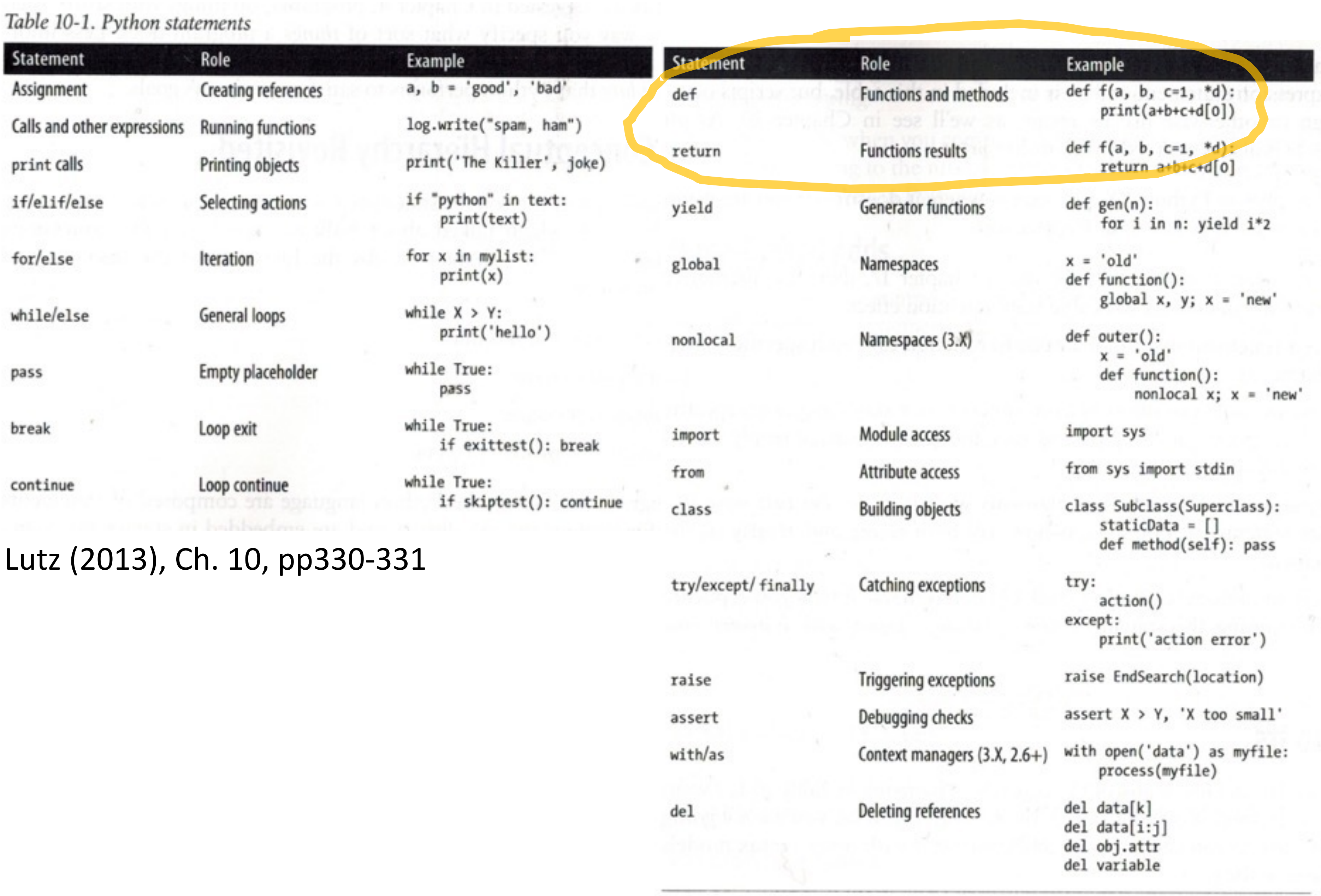


Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	a, b = 'good', 'bad'
Calls and other expressions	Running functions	log.write("spam, ham")
print calls	Printing objects	print('The Killer', joke)

if/elif/else	Selecting actions	if "python" in text:
		    print(text)

for/else	Iteration	for x in mylist: print(x)

while/else	General loops	while X > Y:
		    print('hello')

pass	Empty placeholder	while True:
		    pass

break	Loop exit	while True:
		    if exittest(): break

continue	Loop continue	while True:
		    if skiptest(): continue

def	Functions and methods	def f(a, b, c=1, *d):
		    print(a+b+c+d[0])

return	Functions results	def f(a, b, c=1, *d):
		    return a+b+c+d[0]

yield	Generator functions	def gen(n): for i in n: yield i*2

global	Namespaces	x = 'old'
		def function():
		    global x, y; x = 'new'

nonlocal	Namespaces (3.X)	def outer():
		    x = 'old'
		    def function():
		        nonlocal x; x = 'new'

import	Module access	import sys
from	Attribute access	from sys import stdin

class	Building objects	class Subclass(Superclass):
		    staticData = []
		    def method(self): pass

try/except/ finally	Catching exceptions	try:
		    action()
		except:
		    print('action error')

raise	Triggering exceptions	raise EndSearch(location)
assert	Debugging checks	assert X > Y, 'X too small'
with/as	Context managers (3.X, 2.6+)	with open('data') as myfile: process(myfile)
del	Deleting references	del data[k] del data[i:j] del obj.attr del variable

Lutz (2013), Ch. 10, pp330-331

Built-in functions

abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

Always available built-in functions, descriptions and more:
http://docs.python.org/library/functions.html

Functions

- “Coding an operation as a function makes it a generally useful tool, which we can use in a variety of contexts” (Lutz, p. 473).

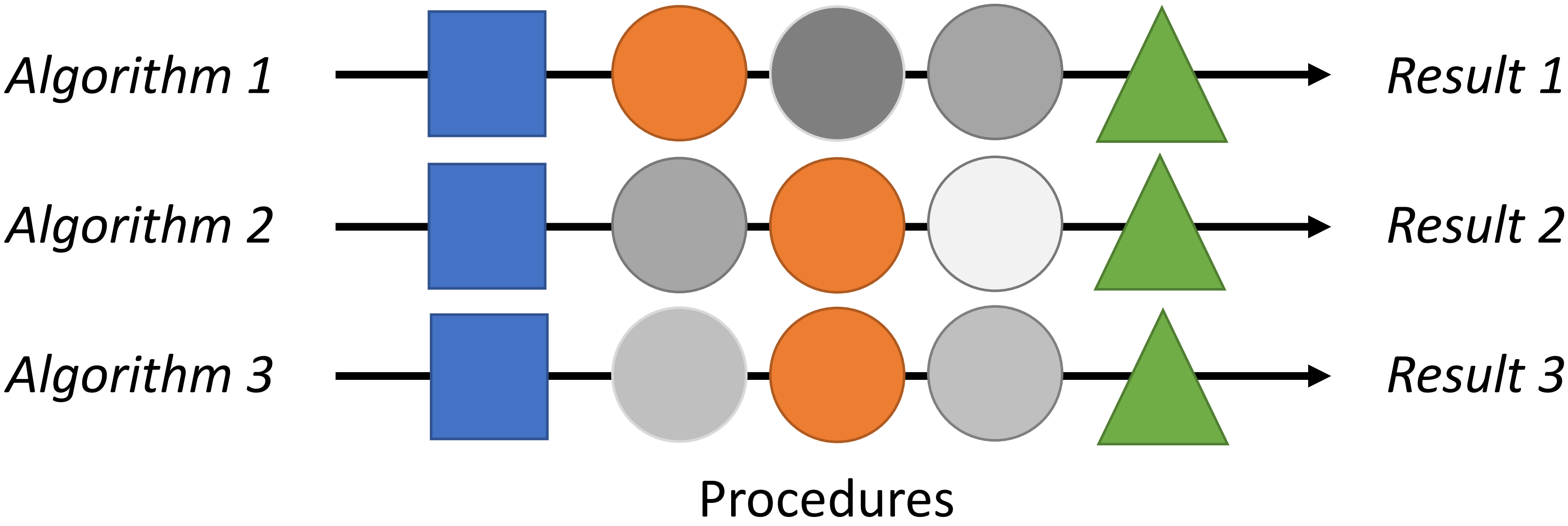
- **packaged procedure invoked by name**.
  = most basic program structure for code reuse
- **housing smaller algorithms, repeatedly used in a program**
- **groups a set of statements**

- Functions also can compute a **result value** and let us **specify parameters** that serve as function inputs and may differ each time the code is run.

Purpose of Functions

Examples for purpose of individual procedures/functions:

- 🟦 Reading specific data format
- ▲ Writing data in another format
- 🟠 Performing data analysis operation used often



Purpose of Functions

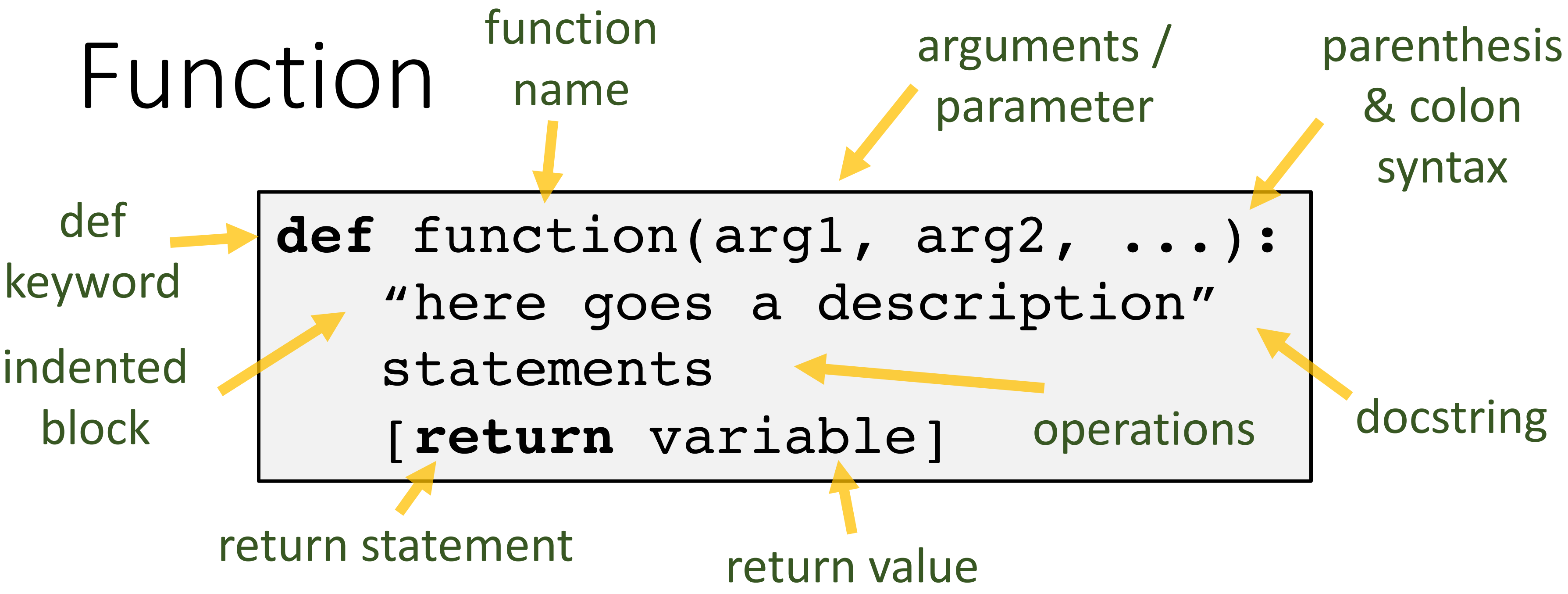
- **Maximize code reuse and minimize redundancy**
  - group and generalize code to be used later
  - factoring tool: repeated use
  - lower redundancy: reduces maintenance effort

- Procedural decomposition
  - splitting systems into pieces
  - chunking tasks
  - implementation of smaller tasks (easier)
  - functions are about procedure: how to do something (rather than what to do)

Concept of Functions

```
functionName(input):
   operation = block of statements
   [return of results]  # optional
```

Function



```
def function(arg1, arg2, ...):
   “here goes a description”
   statements
   [return variable]
```

function name

arguments / parameter

parenthesis & colon syntax

def keyword

indented block

operations

docstring

return statement

return value

- **def** name
  - *def* creates function object and assigns it to a name
  - function name is reference to function object (can be renamed!)
  - def only, content executed only once the function is called
- **agruments/parameter** (optional)
  - passed within parenthesis, by assignment position
- **return** (optional)
  - sends result object back to caller
  - without return statement or return value: returns *None* object

Tutorial

Examples

- Minimum function anatomy
- Calling a function
- Function with return value
- Calling a function with parameter
- Storing the return value in a variable
- Polymorphism
- Type annotations (input and output variable)

Tutorial

Write a function with two input parameter

- Create a new function called `hello` that:
  - receives 2 parameter: `name` of a person and their `age`
  - returns a string greeting the person and informs about the age in 10 years.
- Call the function with appropriate input variables
- Assign function output to a variable called `output`.
- Printing `output` to screen

Where are Functions defined?

Any guess???

- Before you use it !
- First code cells in a notebook, or at the top of a Python script
- In a package / module, which can be imported first!

Tutorial

More Examples

- Default argument

```
def function(arg1, arg2=‘default’)
    statements
    [return variable]
```

- Anonymous Functions

```
lambda arguments: expression
```

- Nested Functions
- Adding a Docstring

Tutorial

Function with Docstring

- Load the python code file 'tempCalculator.py’
- Using the code snippets to create a function `tempCalculator` function that accepts temperatures in Kelvins and returns either Celsius or Fahrenheit, which should be indicated by a parameter.
- Add the provided docstring to the function.
- Test the calculator
- Call the help() function for the calculator.

This will be part of E04

Scopes & Namespaces

Definition

Namespace

- (Program) Space which holds current names of functions or variables
- Place in the program, where names are valid

Namespace of Functions

All variables assigned inside a function are associated with that function's namespace and no other.

By default variables in a *def* are local:

- names assigned inside a *def* only seen by code within that *def*
- names inside a def do not clash with same names outside a def
  - for example: name X assigned outside a *def* is a completely different variable from a name X assigned inside a *def*

Tutorial

Examples

- Current namespace dir()
- Function namespace

Scopes

- Each namespace is assigned a certain scope
  = Area of validity
- Example: function namespace = local scope

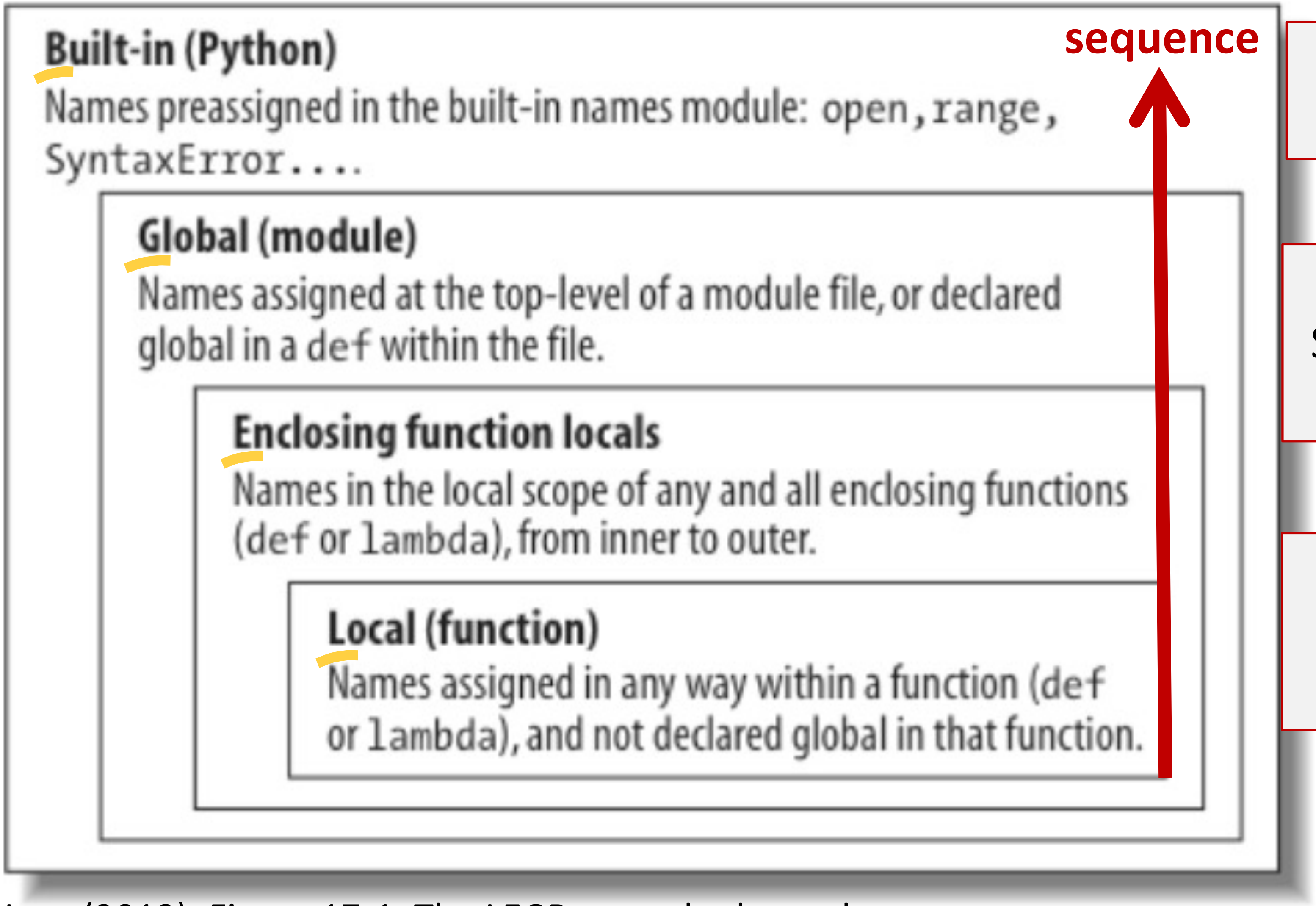
- **Scope = place where variable is existing**
- **Scopes are defined by layered hierarchy**

built-in <- global <- enclosing <- local

Scopes

- **Scope classification at variable assignment:**
  - Location of assignment determines namespace it will live in (=scope of visibility)
  - Assigned names are local, unless declared global
  - **Global scope spans single file only** (more on that later with modules)
  - Each call to a function creates a new local scope
- Scopes help to ...
  - define place where variables are defined & looked up
  - prevent name clashes and interference across your program
  - make functions more self-contained program units (no concern with names used elsewhere)

LEGB Scope Lookup Rule



Name search sequence

Built-in (Python)

Names preassigned in the built-in names module: `open`, `range`, `SyntaxError`....

Global (module)

Names assigned at the top-level of a module file, or declared global in a `def` within the file.

Enclosing function locals

Names in the local scope of any and all enclosing functions (`def` or `lambda`), from inner to outer.

Local (function)

Names assigned in any way within a function (`def` or `lambda`), and not declared global in that function.

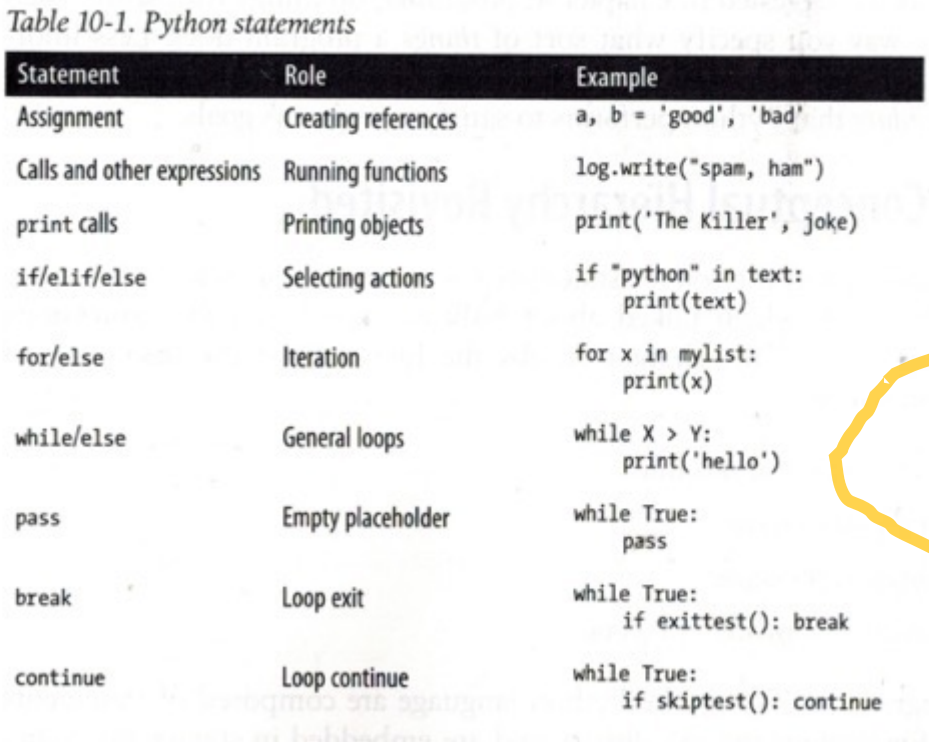
Outer-most

Global: Single File

Local scopes (default)

Lutz (2013), Figure 17-1. The LEGB scope lookup rule

Python Statements



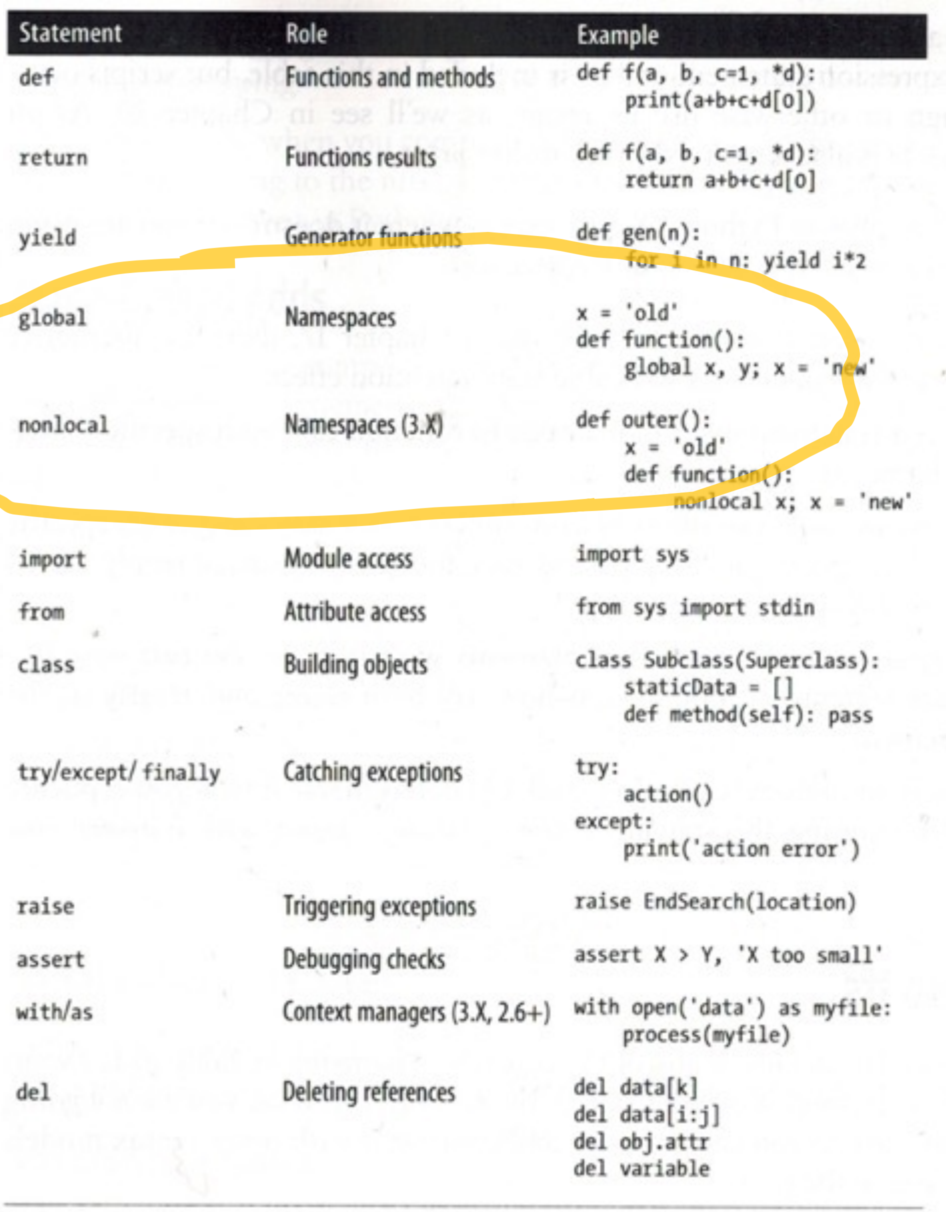


Table 10-1. Python statements

Statement	Role	Example
Assignment	Creating references	`a, b = 'good', 'bad'`
Calls and other expressions	Running functions	`log.write("spam, ham")`
print calls	Printing objects	`print('The Killer', joke)`

if/elif/else	Selecting actions	`if "python" in text:`
		`    print(text)`

for/else	Iteration	`for x in mylist:`
		`    print(x)`

while/else	General loops	`while X > Y:`
		`    print('hello')`

pass	Empty placeholder	`while True:`
		`    pass`

break	Loop exit	`while True:`
		`    if exittest(): break`

continue	Loop continue	`while True:`
		`    if skiptest(): continue`

def	Functions and methods	`def f(a, b, c=1, *d):`
		`    print(a+b+c+d[0])`

return	Functions results	`def f(a, b, c=1, *d):`
		`    return a+b+c+d[0]`

yield	Generator functions	`def gen(n):`
		`    for i in n: yield i*2`

global	Namespaces	`x = 'old'`
		`def function():`
		`    global x, y; x = 'new'`

nonlocal	Namespaces (3.X)	`def outer():`
		`    x = 'old'`
		`    def function():`
		`        nonlocal x; x = 'new'`

import	Module access	`import sys`
from	Attribute access	`from sys import stdin`

class	Building objects	`class Subclass(Superclass):`
		`    staticData = []`
		`    def method(self): pass`

try/except/ finally	Catching exceptions	`try:`
		`    action()`
		`except:`
		`    print('action error')`

raise	Triggering exceptions	`raise EndSearch(location)`
assert	Debugging checks	`assert X > Y, 'X too small'`

with/as	Context managers (3.X, 2.6+)	`with open('data') as myfile:`
		`    process(myfile)`

del	Deleting references	`del data[k]`
		`del data[i:j]`
		`del obj.attr`
		`del variable`

Lutz (2013), Ch. 10, pp330-331

Tutorial

Examples

- Local variables (easier to follow)
- Global variables (better avoided!)
- Nonlocal (very seldomly used):
  - moves from local to enclosing namespace
  - relevant for nested functions

Tutorial

Function with Docstring

- Expand your code for *tempCalculator*
- Add a global variable in the code cell defining freezing temperature in Fahrenheit and Celcius:
  - tempFreezeK = 0
  - Optional: tempFreezeC = 0, tempFreezeF = 32
- Add these global variable as default value of the arguments for functions
  - tempCalculator
  - Optional: celsiustoFahr, kelvinsToCelsius

What is the correct order of arguments in function definition containing default and non-default values

This will be part of E04

Practice

- E04 on Statements and Functions will be available from Friday

  Due Monday 26 September

- Revise L08 notebook on Functions & Scopes
- Optional: L08 Section C on Exceptions