COMP 1510 Winter 2022

Programming Methods

Week 03: Functions, indirection, and memory

Agenda for week 03

1. Programming lifecycle
2. Sequence in Python
3. Truth value testing and selection (branching) in Python using if, if-else, and if-elif-else statements
4. Python keywords and naming conventions
5. Strings in detail
6. ASCII, Unicode, and code points
7. Mutability
8. Dot syntax
9. Basic input; data conversion and str(), int(), float(), etc
10. Formatting output (f-strings, str.format, and %-formatting)
11. Indirection and functions
12. Built-in functions like len()
13. Anatomy of the function
14. Argument passing semantics (pass by value of reference)
15. Functional decomposition
16. First decomposition examples and techniques
17. The main function

PROGRAMMING LIFECYCLE

Recall that a Python program is…

- Sometimes called a script
- A sequence of commands (statements)
- These are evaluated and the statements are executed one at a time in order by the Python interpreter
- We access the Python interpreter using the Python shell
- A program can span multiple files
- Software development and maintenance requires many developers
- How do we organize ourselves?

How do we approach this?

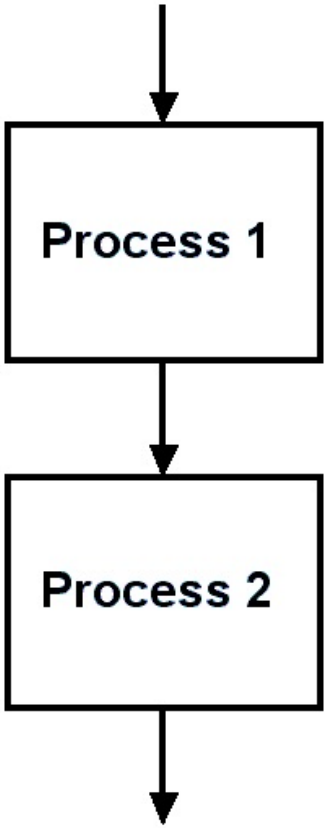
- We use a process
- We call it a **software development life cycle**
- There are many processes:
  - Waterfall
  - Spiral
  - Agile (XP, scrum)
  - Rapid prototyping
  - etc.

- At a programmer level, we:
  1. Plan and design
  2. Use incremental development:
     1. Implement a small change
     2. Ensure it works
     3. Test our implementation
     4. Ensure we haven't broken anything that already exists
     5. Commit to git and push to the cloud
     6. Do it again. And again. And again…!
  3. Debug when required (at every step!)

SEQUENCE

Sequence

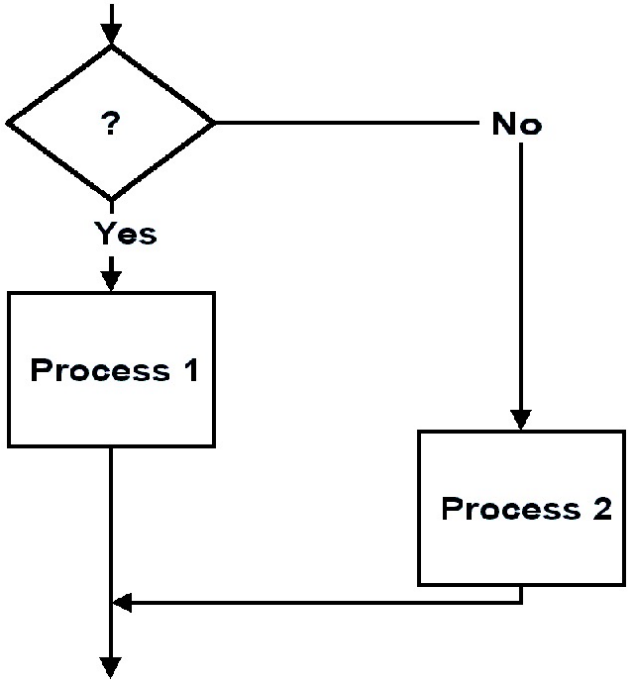
- Top to bottom
- Left to right
- **Ordered statements** executed in order
- Line after line after line after line…
- The interpreter starts at the top…
- …and stops at the end.



SELECTION (BRANCHING)

Selection

- Sometimes we need to **make a choice** that depends on the state of the program
- If X is true, do Y
- If X is true, do Y, else do Z
- If A is less than B, do C
- If T is equal to U, do V, else do W
- Then we continue with the sequence…



Flow of control

- Some programming statements allow us to make decisions and perform repetitions
- These decisions are based on **boolean expressions** (also called **conditions**) that evaluate to True or False
- The sequence and choice of statements being execution is called the flow of control.

Comparison operators

A condition often uses one of Python's **comparison operators** which all return True or False:

Operator	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Some examples

```
a = 4
b = 0
c = (a > b)
print(c) # Prints True

5 == 5
True

0 != 4
True
```

```
‘6.6’ == 6.6
False

‘A’ < ‘B’
True

‘a’ > ‘z’
False

‘hello’ != ‘hi’
True
```

The if statement (in a somewhat complex way)

```
if_stmt ::=  "if" expression ":" suite
             ("elif" expression ":" suite)*
             ["else" ":" suite]
```

An example makes it much easier!

```
password = input("Enter your password: ")

if password == "1234":
    print("Unacceptable password")
else:
    print("Thank you")

print("End of program")
```

Branching: if

- A branch in a program is *only taken if an expression's value is True*
- This is known as an if-branch:

```
if earned_bonus == True:
    salary = salary * 1.2

… and then we continue doing stuff…
```

Branching: if-else

An if-else structure has two branches:

1. The first branch is executed if the testing expression is true
2. The second branch is executed if the testing expression is false

```
if number < 0:
    positive = False
else:
    positive = True
```

Another example

Getting the maximum of two numbers:

```
if first >= second:
    max_value = first
else
    max_value = second
```

Be careful with your indentation!

We can insert a *group (block) of statements* after if and else:

```
if number < 0:
    print(”Negative”)
    print(”Yay”)
else:
    print(”Positive”)
    print(”Boo”)
```

Self-check: can you describe this code?

```
hotel_rate = 155
user_age = int(input('Enter age: '))
if user_age > 65:
    print(“We must honour the elderly”)
    hotel_rate = hotel_rate - 20
print('Your hotel rate:', hotel_rate)
```

Self-check: How much do you pay?

```
user_age = int(input('Enter age: '))

if user_age < 25:
    insurance_price = 4800
else:
    insurance_price = 2200

print('Annual price: $%d' % insurance_price)
```

Self-check

Suppose variables population and land_area_km refer to floating point numbers:

1. Write an if statement that will print the population if it is less than 10_000_000.
2. Write an if statement that will print "Densely populated" if the land density (number of people per unit area) is greater than 60.0 people per km^2.
3. Write an if statement that will print "Densely populated" if the land density (number of people per unit area) is greater than 60.0 people per km^2 and "Sparsely populated" otherwise.

Multi-branch if-else

```
if expression1:
    # Statements that execute when expression1 is true
    # (first branch)
elif expression2:
    # Statements that execute when expression1 is false
    # and expression2 is true
    # (second branch)
else:
    # Statements that execute when expression1 is false
    # and expression2 is false
    # (third branch)
```

Another example

```
password = input("Enter your password: ")

if password == "1234":
     print("Unacceptable password")
elif password == "abcd":
     print("That's a terrible password leave now")
else:
     print("Thank you")

print("End of program")
```

Example

```
num_years = int(input('Enter number years married: '))

if num_years == 1:
   print('Your first year -- great!')
elif num_years == 10:
   print('A whole decade -- impressive.')
elif num_years == 25:
   print('Your silver anniversary -- enjoy.')
elif num_years == 50:
   print('Your golden anniversary -- amazing.')
else:
   print('Nothing special.')
```

Self-check: translate these into Python

Create this decision structure:

a) If year is 2101 or later, print "In the distant future" (without quotes)
b) Otherwise, if year is 2001 or greater, print "21st century"
c) Otherwise, if year is 1901 or greater, print "20th century"
d) Else (1900 or earlier), print "Long, long ago…".

Nested if-else statements

A branch's statements can include any valid statements, including another if-else statement:

```
if pH < 7.0:
   print("That's acidic!")
   if pH < 3:
       print("Don't drink that")
   else:
       print("I love vinegar!")
else:
   print("That's alkaline!")
```

We can use multiple sequential ifs

- Sometimes we can use multiple if statements in sequence
- *Each if-statement is independent*
- More than one may end up executing:

```
user_age = int(input('Enter age: '))
if user_age >= 16:
   print('You are old enough to drive.')
if user_age < 25:
   print('Enjoy your early years.')
if user_age > 25:
   print('Most car rental companies will rent to you.')
```

NAMING CONVENTIONS

Can I use any word for a variable name?

- **No!**
- There are special words in the Python language
- These **keywords** mean something specific and cannot be used for identifiers
- Keywords are part of the language
- *There are 35 keywords in Python*
- You must learn them all!

Python keywords (a reminder!)

```
False      await      else       import     pass
True       break      except     in         raise
None       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

Are there any naming conventions?

Yes! Lots!

- We have **conventions** and **good practices** in programming
- Generally accepted **standards** we all aim to meet
- Naming things in our programs is described by **PEP 8**
- PEP: Python Enhancement Proposal
- PEP 8: style guide for formatting Python code
- Readme: https://www.python.org/dev/peps/pep-0008/#descriptive-naming-styles

PEP 8 summary

1. Identifiers can only contain letters, numbers, and underscores
2. Spaces are not allowed in variable names
3. Avoid using Python keywords and function names, i.e., print
4. Identifiers should be short but descriptive
5. Don't confuse the lower case letter l with the number 1
6. Don't confuse the upper case O with the number 0
7. Variables should be in lowercase_underscore format (🐍-case)
8. Names that start and end with a double underscore should be avoided (Python does something special with these wait and see…)

What drives naming conventions?

GOAL:

- We want our code to be easy to read and understand
- It should be a quick skim and not an intense study!

HOW:

- *Use the paperback model* – the author is responsible for making themself clear
- *Do not use the academic model* where it is the scholar's job to dig the meaning out of the paper

Some guidelines…

1. Use intention-revealing names
    1. If a name requires a comment, then the name does not reveal its intent
    2. Why, what, and how
2. Avoid disinformation
    1. Don't vary from entrenched or common meanings of words
    2. Don’t use very similar names
3. Make meaningful distinctions
    1. No noise words (no redundant words, be precise!)
    2. No number-series naming, i.e., input_1, input_2 – don’t do this!
4. Use pronounceable names

Some guidelines…

5. Use searchable names
    1. No single-letter names
    2. The length of a name should generally correspond to its importance
6. Avoid encodings
    1. No Hungarian notation (don't encode the type in the name)
    2. No member prefixes like m_
7. Avoid mental mapping
    1. Clarity is king
    2. Avoid single letter variable names, i.e., if you can reliably remember that r is the lower-cased version of the URL with the host and scheme removed, then you must clearly be very smart wow

Some guidelines…

8. Don't be cute – choose clarity over entertainment value
9. Pick one word per concept, i.e., don't use fetch, get, and retrieve just pick one and be consistent in your application
10. Don't pun, i.e., don't use the same word for two purposes
11. Use solution domain names (your audience is other programmers, so use computing terms)
12. Use problem domain names
13. Don't add gratuitous content. Be specific and succinct!

STRINGS IN DETAIL

The Python string (str)

- A *string* is an immutable sequence (or string) of Unicode (UTF-16) codepoints
- We can bind a string object to a variable
- A *string literal* is the value of a string
- We create a string literal using double- **or** single-quotes
- May contain letters, numbers, spaces, or symbols like @ or #.

```
"BCIT CST grads are the best!"
'BCIT CST grads make the most money!'
```

A string is a “sequence” type

- A string’s characters are kept in order from first to last
- A character’s position in a string is called its index
- The plural of index is *indices*
- Indices start at 0

P	Y	T	H	O	N
0	1	2	3	4	5

- An empty Python string has zero elements and is created like this:

```
my_empty_string = ""
```

ASCII, UNICODE, AND CODE POINTS

ASCII TABLE

https://upload.wikimedia.org/wikipedia/commons/1/1b/ASCII-Table-wide.svg

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	*[NULL]*	32	20	*[SPACE]*	64	40	@	96	60	`
1	1	*[START OF HEADING]*	33	21	!	65	41	A	97	61	a
2	2	*[START OF TEXT]*	34	22	"	66	42	B	98	62	b
3	3	*[END OF TEXT]*	35	23	#	67	43	C	99	63	c
4	4	*[END OF TRANSMISSION]*	36	24	$	68	44	D	100	64	d
5	5	*[ENQUIRY]*	37	25	%	69	45	E	101	65	e
6	6	*[ACKNOWLEDGE]*	38	26	&	70	46	F	102	66	f
7	7	*[BELL]*	39	27	'	71	47	G	103	67	g
8	8	*[BACKSPACE]*	40	28	(	72	48	H	104	68	h
9	9	*[HORIZONTAL TAB]*	41	29	)	73	49	I	105	69	i
10	A	*[LINE FEED]*	42	2A	*	74	4A	J	106	6A	j
11	B	*[VERTICAL TAB]*	43	2B	+	75	4B	K	107	6B	k
12	C	*[FORM FEED]*	44	2C	,	76	4C	L	108	6C	l
13	D	*[CARRIAGE RETURN]*	45	2D	-	77	4D	M	109	6D	m
14	E	*[SHIFT OUT]*	46	2E	.	78	4E	N	110	6E	n
15	F	*[SHIFT IN]*	47	2F	/	79	4F	O	111	6F	o
16	10	*[DATA LINK ESCAPE]*	48	30	0	80	50	P	112	70	p
17	11	*[DEVICE CONTROL 1]*	49	31	1	81	51	Q	113	71	q
18	12	*[DEVICE CONTROL 2]*	50	32	2	82	52	R	114	72	r
19	13	*[DEVICE CONTROL 3]*	51	33	3	83	53	S	115	73	s
20	14	*[DEVICE CONTROL 4]*	52	34	4	84	54	T	116	74	t
21	15	*[NEGATIVE ACKNOWLEDGE]*	53	35	5	85	55	U	117	75	u
22	16	*[SYNCHRONOUS IDLE]*	54	36	6	86	56	V	118	76	v
23	17	*[ENG OF TRANS. BLOCK]*	55	37	7	87	57	W	119	77	w
24	18	*[CANCEL]*	56	38	8	88	58	X	120	78	x
25	19	*[END OF MEDIUM]*	57	39	9	89	59	Y	121	79	y
26	1A	*[SUBSTITUTE]*	58	3A	:	90	5A	Z	122	7A	z
27	1B	*[ESCAPE]*	59	3B	;	91	5B	[	123	7B	{
28	1C	*[FILE SEPARATOR]*	60	3C	<	92	5C	\	124	7C	|
29	1D	*[GROUP SEPARATOR]*	61	3D	=	93	5D	]	125	7D	}
30	1E	*[RECORD SEPARATOR]*	62	3E	>	94	5E	^	126	7E	~
31	1F	*[UNIT SEPARATOR]*	63	3F	?	95	5F	_	127	7F	*[DEL]*

- Programmers used to use ASCII (American Standard Code for Information Interchange)
- ASCII encodes 128 characters, mostly from English
- This was marginally acceptable in the early days of computing, but 128 characters is obviously insufficient now
- Unicode is a superset of ASCII (the first 128 characters of Unicode are the ASCII characters)
- Python uses Unicode to represent every possible character as a unique integer, known as a code point
- For example, the character 'A' has the code point value of 65
- The built-in function ord( ) returns the encoded code point integer for a string of length one (a Unicode character).
- The built-in function chr( ) returns the one-character string for a code point passed as an argument.

ESCAPE CHARACTERS

In order to represent quotes inside a string or characters with special meaning, we preface them with the back-slash \

Escape Sequence	Meaning
\’	Single quote (‘)
\”	Double quote (“)
\t	Horizontal tab
\n	New line (line feed)
\b	Backspace
\r	Carriage return

Raw strings

- If we do not want to use escape characters, we can create a raw string
- A raw string is prefixed by the letter r or R
- Python raw strings treat the backslash as a literal character
- This is useful if we have a string that contains backslashes that should be interpreted as backslashes
- Compare:

```
1. print("Hello\nworld")
2. print(r"Hello\nworld") # This is a raw string
```

INTRODUCING DOT SYNTAX

What can we do with strings?

- We can change the case by accessing a *method* called title( )

```
word = 'supercalifragilisticexpialidocious'
print(word.title());
```

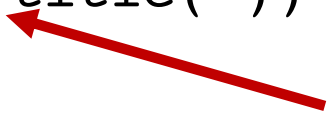
- A method is a special function that Python can invoke "on an object" using dot syntax
- A method is followed by a set of parentheses which often contains additional information needed by the method
- The title( ) method does not require additional information, but we still use the parentheses because it's a method.

Dot syntax

We see this in nearly all modern programming languages:

1. We have some sort of object
2. We want to ask the object to do something
3. We use dot syntax to invoke the behaviour of the object

```
word = 'supercalifragilisticexpialidocious'
print(word.title( ))
```



Dot syntax to invoke a function (method)

What else can we do?

- We can invoke a method called upper( ) to acquire an upper-case copy of the string
- We can invoke a method called lower( ) to acquire a lower-case copy of the string

- *Question*: if we invoke these methods, does the original string change?
- *Question*: How can we test this?

INTRODUCING MUTABILITY

im·mu·ta·ble

/i(m)ˈmyo͞odəb(ə)l/

Adjective

- From the Latin roots “in” (not) and “mutabilis” (to change)
- unchanging over time or unable to be changed

Python strings are immutable

- Writing or altering individual characters of a string object is not allowed
- *Strings cannot change once created*
- Instead, an assignment statement must be used

```
word = 'supercalifragilisticexpialidocious'
new_word = word.title( )
print(word)
print(new_word)
```

There are a few immutable types in Python

- int
- float
- str
- tuple
- complex
- bytes
- frozenset
- Everything else is mutable (I think!).

Assignment is not mutation in Python

When we assign a value to a variable, we are binding a location in memory where we store that value to the variable (identifier)

```
test_value = 2.5
type(test_value)
id(test_value)
test_value = test_value + 0.0456
id(test_value) # test_value stores a different address!
```

Back to strings… a self-check!

Write a snippet of code to prove you can use each of these string methods:

1. capitalize
2. replace
3. isalpha
4. isdigit
5. split
6. join

How can we find out a string's length?

We can use the global len( ) function and pass it the name of the string:

```
word = 'supercalifragilisticexpialidocious'
number_of_characters = len(word)
```

How can we acquire a character*?

*A string of length 1

- Use ***square brackets*** and the index
- Remember the index starts at 0

```
word = 'supercalifragilisticexpialidocious'
first_letter = word[0]
second_letter = word[1]
third_letter = word[2]
final_letter = word[-1] # Neat!
penultimate_letter = word[-2] # Super neat!
```

Can we combine strings?

- *Yes!* We call this **concatenation**
- String concatenation is performed using the + operator in Python
- It results in a new string (the originals are unchanged)
- We need to assign the new string to a variable, or it is deleted from memory immediately after it is created (poof!):

```
greeting = "hello "
guest = "world"
welcome_statement = greeting + guest
```

Concatenating strings with non-strings

What is this madness?

```
>>> name = ‘Chris'
>>> age = 29 #hahahaha
>>> surname = ‘Thompson’
>>> message = name + ‘ ’ + surname + ‘ is ’ + age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Concatenating strings with non-strings

That's better

```
>>> name = 'Chris'
>>> age = 48 # Sigh
>>> surname = 'Thompson'
>>> message = name + ' ' + surname + ' is ' + str(age)
```

We *can* multiply a string by an int

```
>>> best_school = 'BCIT'
>>> cheer = best_school * 4
>>> print(cheer)
'BCITBCITBCITBCIT'
```

Question: will best_school * best_school work?

Multi-line strings

There are three ways we can do this:

1. Break lines with the \ character:

```
string_long = "This is a very long string" \
              " that I wrote to help somebody" \
              " who had a question about" \
              " writing long strings in Python"
```

Multi-line strings

2. Use parentheses:

```
string_long = ("This is a very long string"
               " that I wrote to help somebody"
               " who had a question about”
               " writing long strings in Python”)
```

Multi-line strings

3. Use triple quotes:

```
string_long = """This is a very long string
                 that I wrote to help somebody
                 who had a question about
                 writing long strings in Python"""
```

What about whitespace?

- When users enter information, they often enter unnecessary whitespace
- Python has some built-in methods to remove this whitespace:

```
another_string = '     hello world     '
another_string.strip( ) # strips all whitespace
another_string.lstrip( ) # strips leading whitespace
another_string.rstrip( ) # strips trailing whitespace
```

Is that everything about strings & mutability?

- **No!**
- We will learn more about strings as we proceed through the term
- You can now **store** strings in variables, **concatenate** strings, **modify** strings using string methods, and **convert** strings to numbers and vice versa

FORMATTING OUTPUT

Different versions of print( )

There are different versions of the print( ) function:

```
1. print('Hello world')
2. print("Hello world")
3. print("Hello world", end=' ')
4. print("I\'m", age, "years old")
```

* Actually concatenation first

Formatting output

There are three easy ways to format output in Python:

1. the string format method
2. f-strings
3. format specifiers

The str.format( ) method

- The format method inserts values into strings
- Replacement fields are delimited by curly braces:

```
"I earned {0} in {1}".format(98.5, "Architecture")
"I earned {} in {}".format(98.5, "Architecture")
"I earned {0} in {1} and {0} in {2}".format(65, "Comm")
"I earned {grade} in {course}".format(grade=65, course="Comm")
grades = (98.5, 65)
"I earned {g[0]} and {g[1]}".format(g = grades)
import math
"Pi equals {m.pi}".format(m = math)
"Pi equals {m.pi:.3f}".format(m = math)
```

f-strings

Read more here: https://www.python.org/dev/peps/pep-0498/

- f-strings are string literals preceded by an f or F
- Curly braces contain expressions that can be replaced with their values
- The f in f-string stands for fast – this is faster than str's format method
- Evaluated at runtime:

```
grade = 98.5
course = "C"
f"I earned {grade} in {course}"
name = "OOP 2 with Python"
F"My favourite course (so far) is {name.title()}!"
```

Python's print can use conversion specifiers

- Program output commonly includes the value of variables as a part of the text
- A **string formatting expression** allows a programmer to create a string with placeholders that are replaced by the value of variables
- This placeholder is called a **conversion specifier**
- Different conversion specifiers are used to perform a conversion of a given variable value to a different type when creating a string

Common conversion specifiers

Conversion Specifier	Notes	Example	Output
%d	For a decimal integer	print('%d' % 17)	17
%f	For a floating point number	print('%f' % 3.14)	3.14
%s	For a string	print('%s' % name)	Oliver
%x, %X	For hexadecimal	print('%X' % 17)	11
%e, %E	For scientific notation	print('%e' % 314)	3.140000e+02

What if we want to print the %

```
apr = float(input('Enter APR:\n'))

# Print using a float conversion specifier
print('Annual percentage rate as a float is %f ' % apr)

# Print using a float conversion specifier and trailing %
print('Annual percentage rate as a float is %f%% ' % apr)
```

What if we want to print more than one?

Super easy:

```
name = ‘COMP 1510’
room = 665
print(‘Course %s is in %d\n’ % (name, room))
```

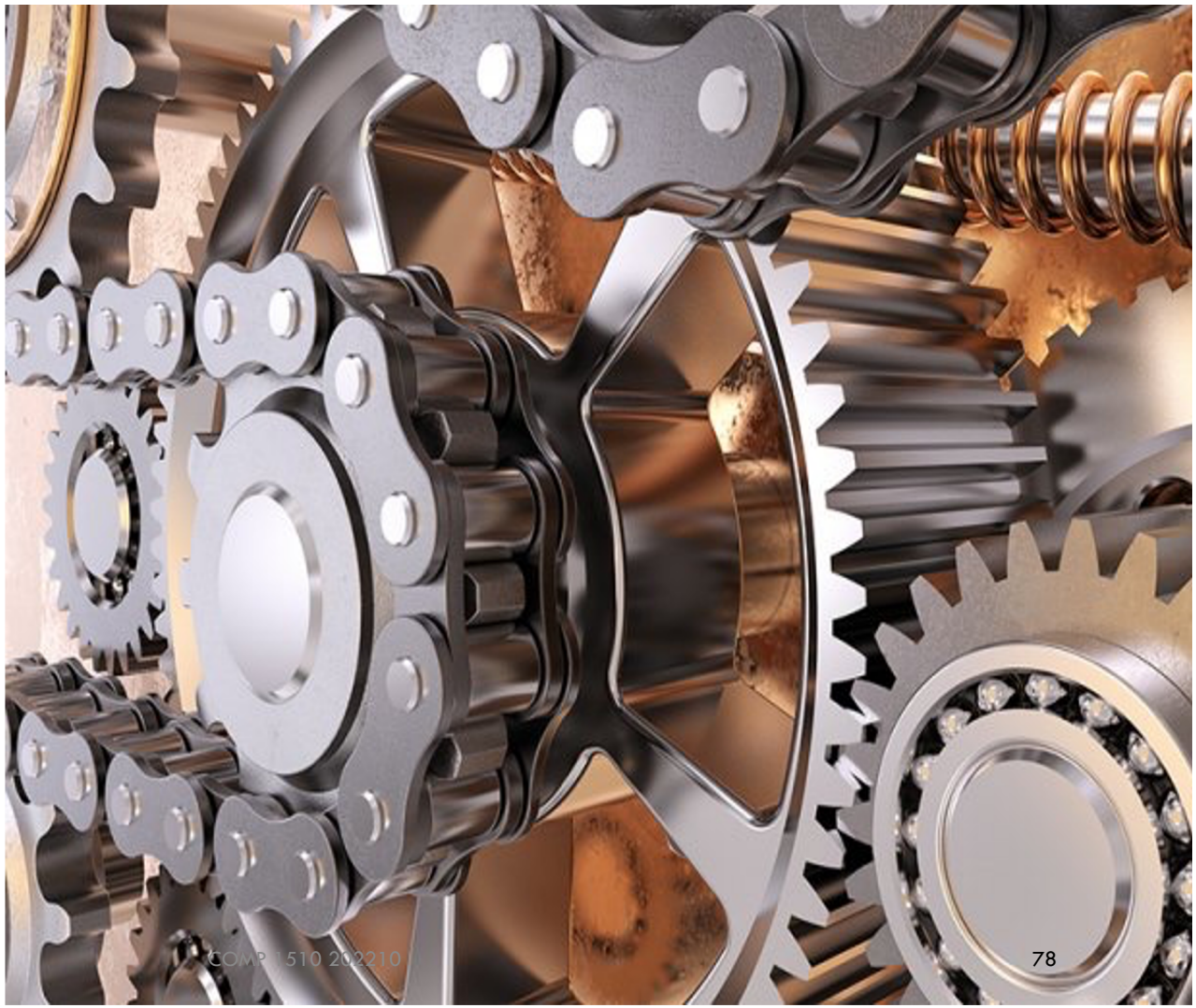
INDIRECTION AND FUNCTIONS

Motivation

- Writing code in one long sequence is not a good idea
- Difficult to read
- Difficult to maintain
- Instead we **employ indirection** and organize our code into modules
- The smallest module is the **function**.

Which functions have we used?

- We've already met some of Python's built-in functions:
  - print( )
  - input( )
  - int( )
  - float( )
  - str( )
  - len(
  - quit( )
  - ord( )
  - chr( )

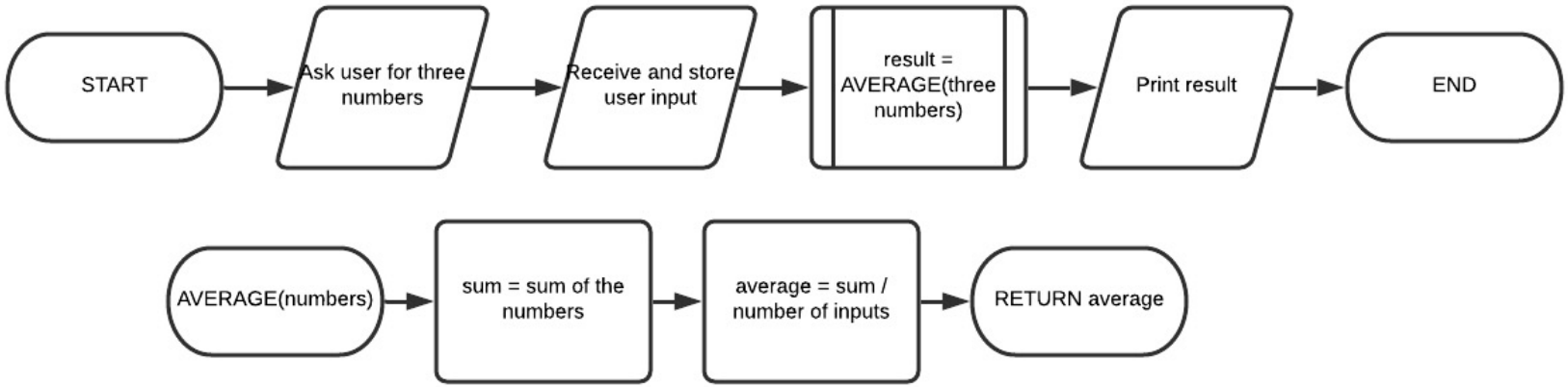


INDIRECTION

Indirection

- In order to write serious programs, we need to modularize our code
- Scripts are fine for DevOps
- Scripts are not fine for programming
- We will begin writing code in named, re-usable blocks called functions
- We will store groups of related functions in source files
- We will call these source files modules
- We will write programs that import code from other modules.

Flowchart example: indirection



ANATOMY OF THE FUNCTION

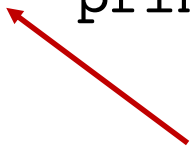
We can write our own functions too

A function definition consists of:

1. The keyword def
2. The new function's name followed by parentheses and a colon
3. An indented **block** of statements

```
def print_greeting( ):
    print('Hello there!')
```

Notice the indentation? It's *mandatory!*



1 tab aka **4** spaces

Anatomy of a function

Function header or signature

```
def print_greeting( ):
    print('Hello there!')
```

Notice the indentation? It's mandatory!

Function body or implementation

https://www.etsy.com/ca/shop/aKNITomy

Some functions require information

- When a function requires information, we say it *accepts* ***parameters***
- We must pass values to such a function for it to work
- We call the values that we pass to the function ***arguments***
- Every function call can be made many times with a different (or the same) argument each time

```
def print_greeting(name):
    print('Hello ' + name)
```

Some functions require lots of info

- Functions accept a ***comma-separated list of parameters***
- Order matters (for now!)
- **ALL** arguments must be provided in the order listed by the parameters

```
def divide(dividend, divisor):
    quotient = dividend / divisor
    return quotient
```

Functions can return the values too

- We ***use a return statement*** to take the value from the function and return its address to the calling code
- A function can return one value

```
def add(int_1, int_2):
    """Return the sum of the two parameters"""
    sum = int_1 + int_2
    return sum
```

I'd use our add function like this:

```
a = 10
b = 20
c = add(a, b)

# will print the value in c, which is 30
print(c)
```

A function may call a function may call…

- A function's statements may include function calls
- A function may call a function or several functions
- These are known as ***hierarchical function calls*** or ***nested function calls***
- Programmers like to do this
- Functions should be very short and atomic
- Functions should do one thing
- Just one thing
- If a function does more than one thing, it should be "decomposed" into two or more functions

Here's a function calling a function:

```
user_input = int( input( ) )
```

This statement consists of such a hierarchical function call:

1. the input( ) function is called and evaluates to a value
2. The value returned from the input( ) function is IMMEDIATELY passed as an argument to the int( ) function
3. The int function converts (hopefully) the input into an integer, which is put into a new object in memory and bound to the variable user_input.

Here's another example

```
a = 5
b = 4
c = -3
result = sum_of_square(a, b, c)
print(result)
```

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

def sum_of_square(x, y, z):
      xx = square(x)
      yy = square(y)
      zz = square(z)
      return xx + yy + zz
```

Why use functions?

- Improve program readability
- Encourages modular program development
- Reduce complexity
- Minimizes redundant code.

Those are all the reasons. But that's a lot. Functions make coding much easier.

Does order matter? Will this work?

```
# I'm going to invoke (use) a function
my_function( )

# And define the function AFTER I invoke it.
def my_function( ):
    print("I defined this function AFTER I invoked it")
```

NO THIS WILL NOT WORK!

The interpreter reads the file top down!

ARGUMENT PASSING SEMANTICS

How do we provide data to functions?

- When we "pass" a variable to a function, exactly what are we giving to the function?
- In programming we generally have two choices:
  1. Pass by reference
  2. Pass by value
- ***Python is pass by value (of reference)***
- When we pass something to a function, we pass the value in the variable
- Python variables contain references, so we pass a number aka an address!

The quintessential swap example

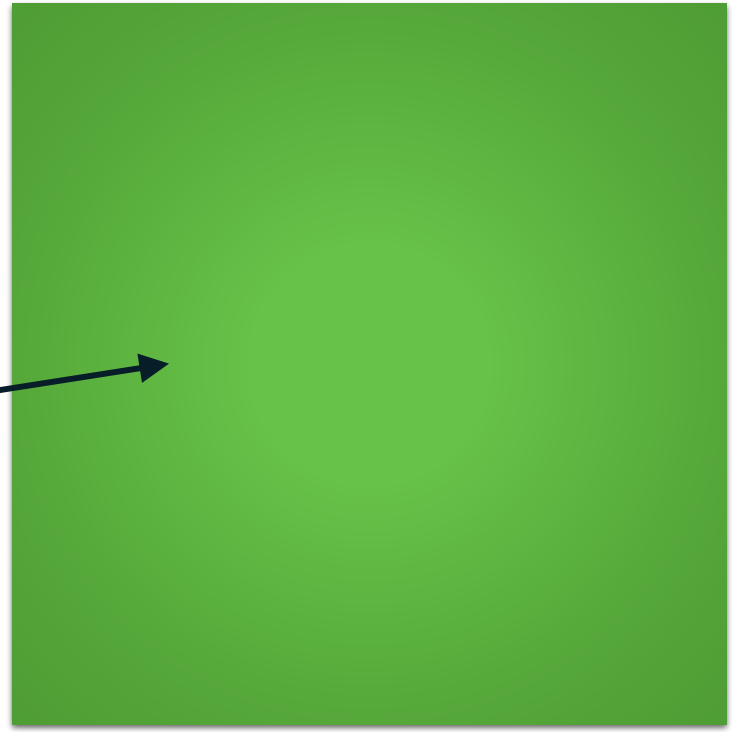
What does this function do?

```
def swap(a, b):
    temp = a
    a = b
    b = temp
```

Let's create a memory model

```
def swap(a, b):
    temp = a
    a = b
    b = temp
```

1. Let's pretend this is the memory heap
2. When we instantiate objects they are created inside the heap

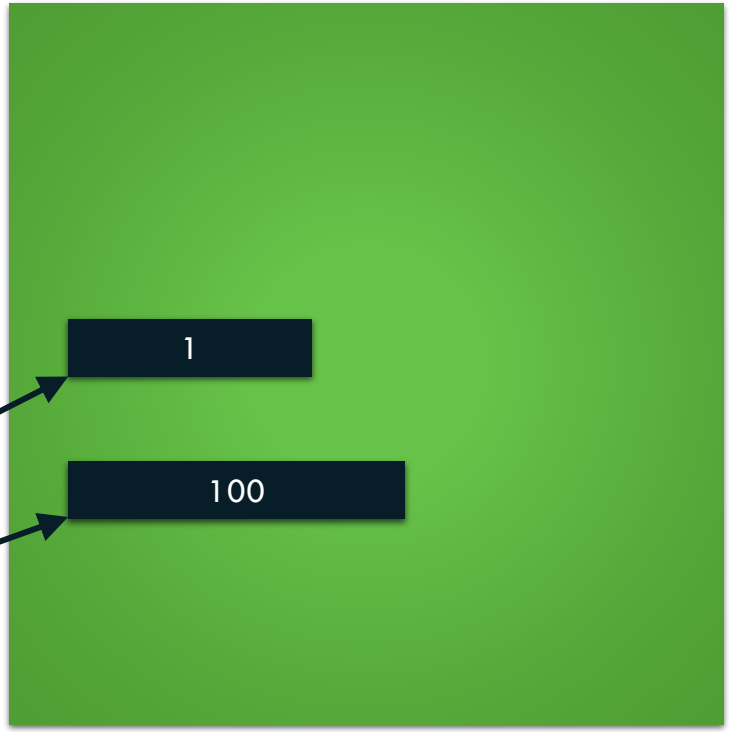


Let's create a memory model

```
def swap(a, b):
    temp = a
    a = b
    b = temp
```

3. Now let's instantiate two objects:

```
first = 1
last = 100
```

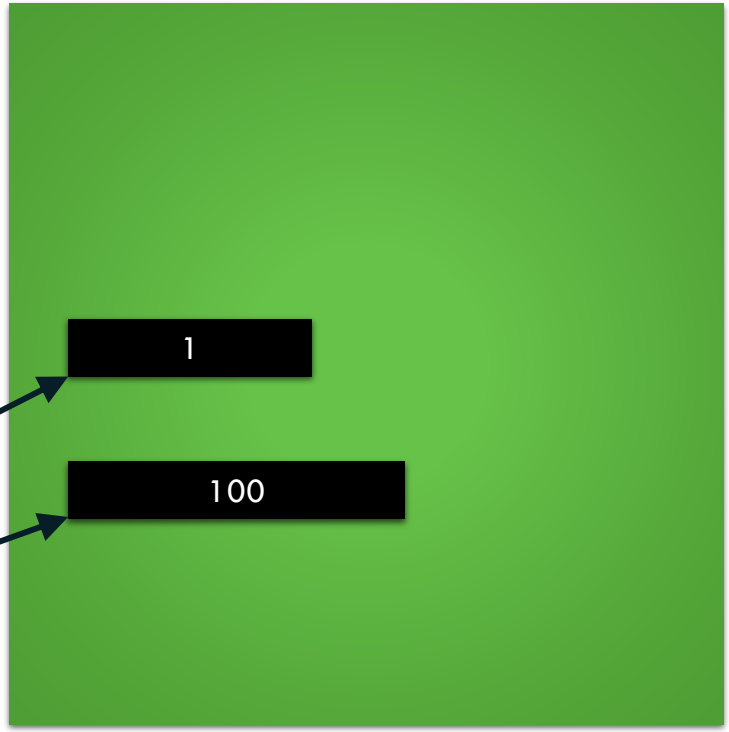


Let's create a memory model

```
def swap(a, b):
    temp = a
    a = b
    b = temp
```

4. Our variables contain the addresses of the objects (like C pointers):

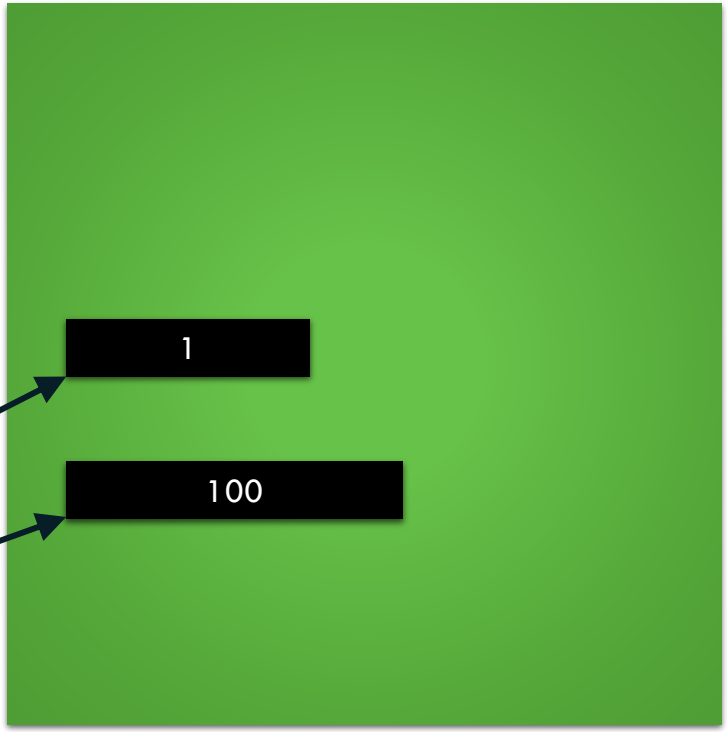
```
first = 1
last = 100
```



Let's pass our objects to the swap

```
def swap(a, b):
    temp = a
    a = b
    b = temp
```

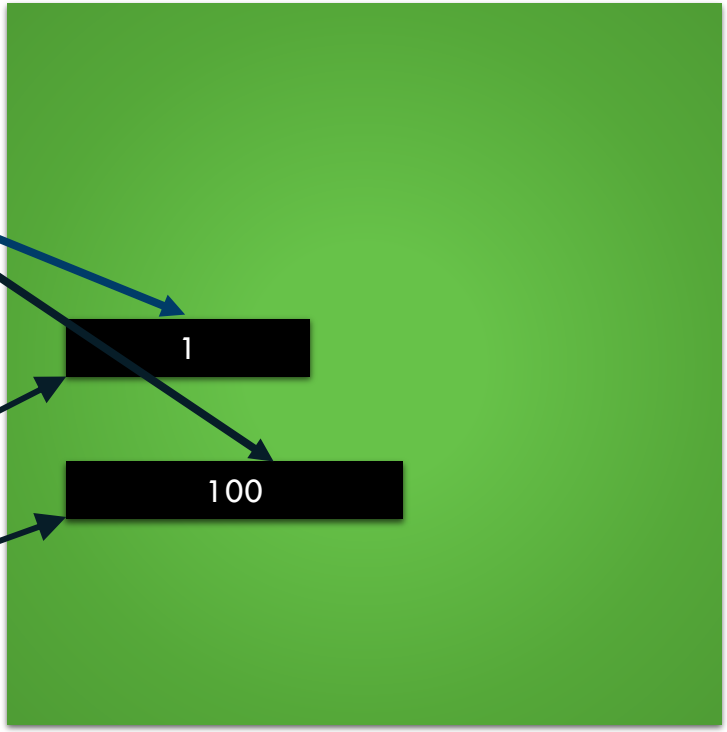
```
swap(first, last)
```




```
first = 1
last = 100
```

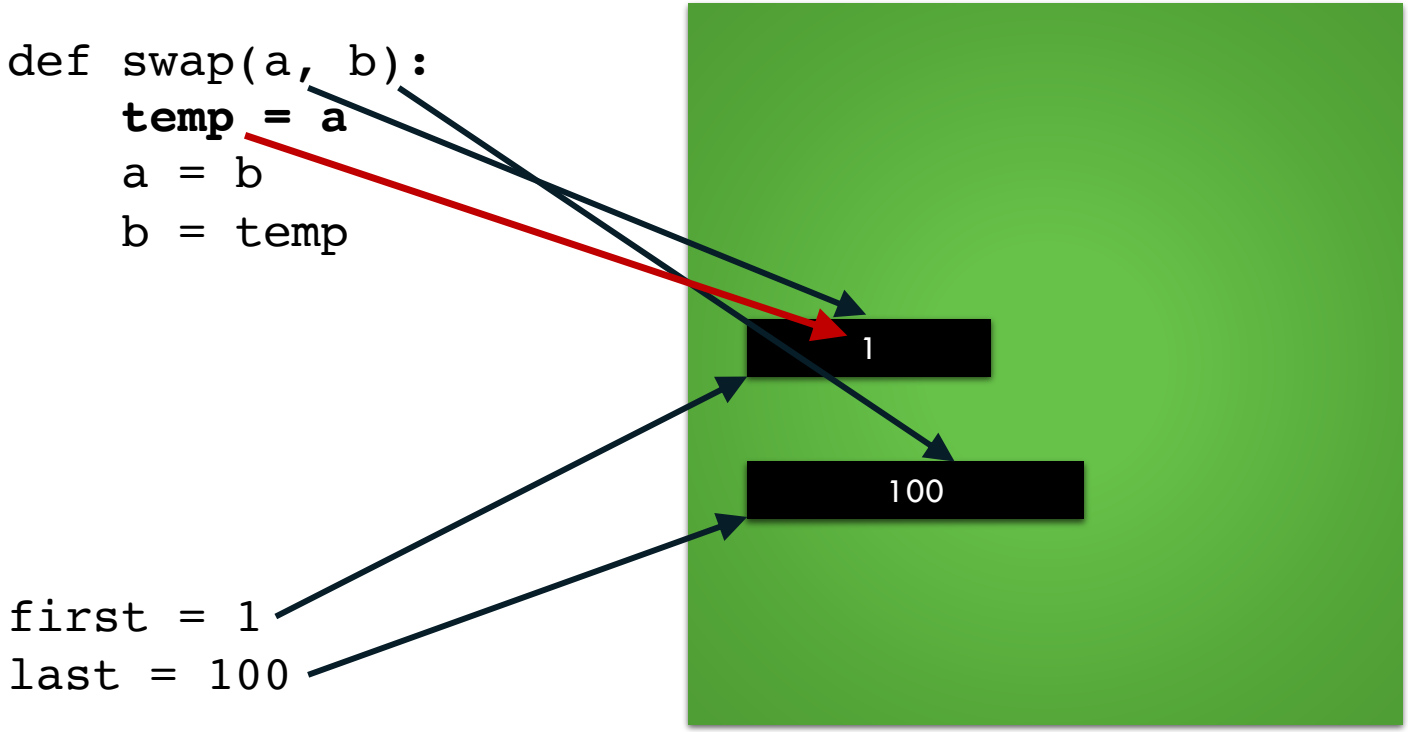
1. Pass copies of the values of the references as the parameters

```
def swap(a, b):
    temp = a
    a = b
    b = temp
```

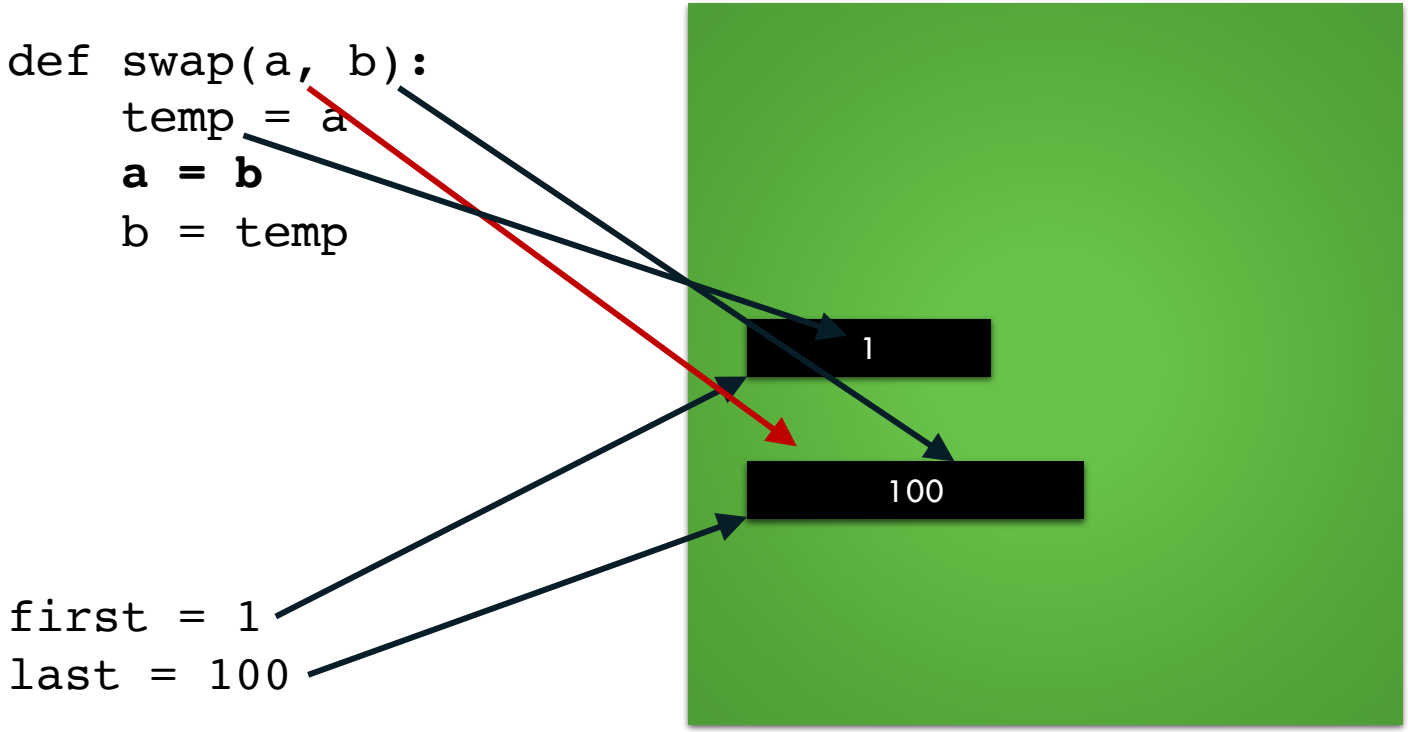



```
first = 1
last = 100
```

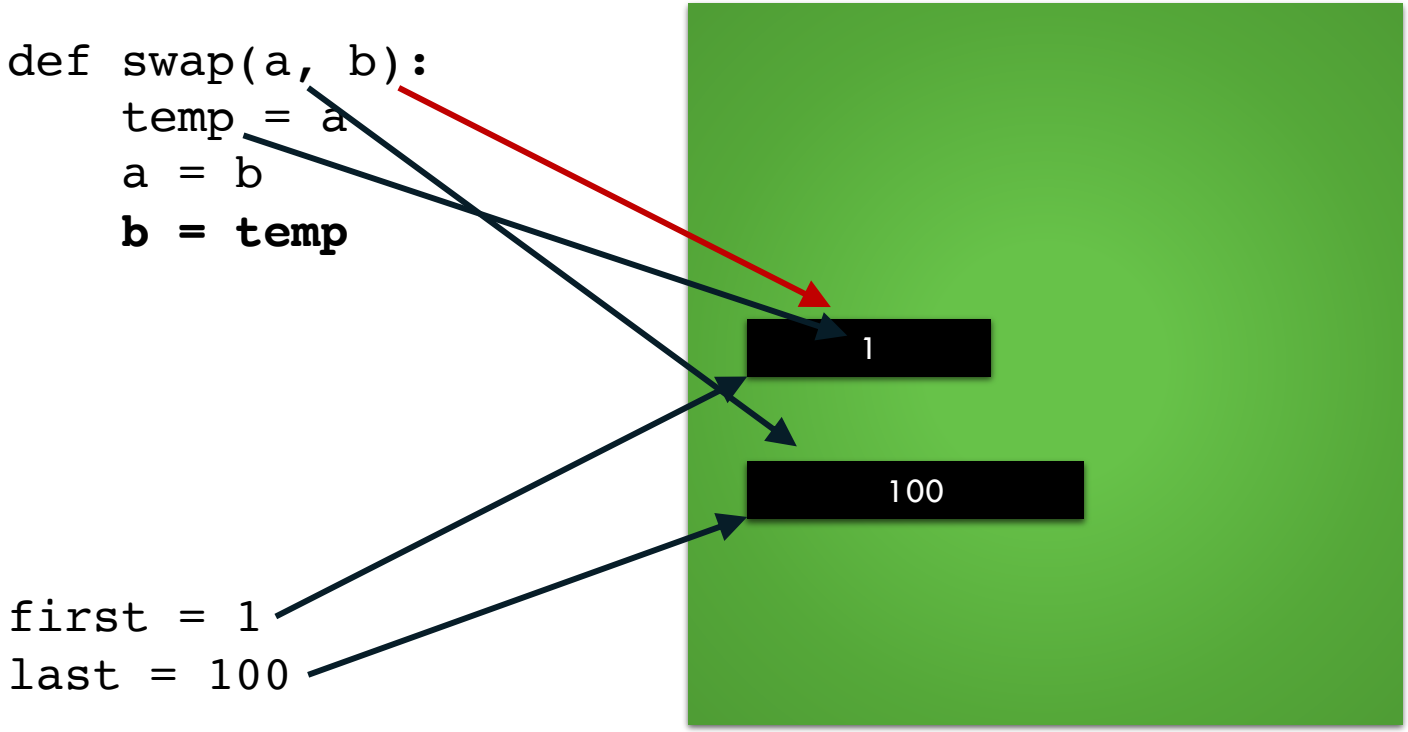
2. Execute the code (line 1)



2. Execute the code (line 2)



3. Execute the code (line 3)

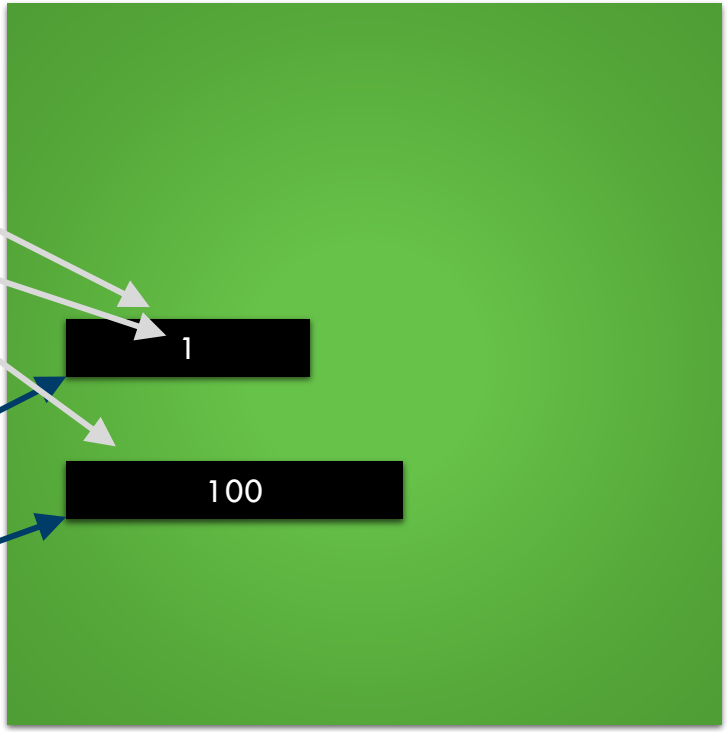


4. Exit the function

```
def swap(a, b):
    temp = a
    a = b
    b = temp
```

The original references are untouched!

```
first = 1
last = 100
```



The swap method did nothing

1. We copied the value of the variables and passed the copies as the arguments (parameters) of the swap method
2. In Python we always copy the address of the object in memory
3. We messed around with the parameters and the addresses they contain
4. The original variables and their values (addresses) remained untouched

CONCLUSION: This swap doesn't work in Python

CONCLUSION: Python is PASS BY VALUE OF REFERENCE!

FUNCTIONAL DECOMPOSITION

Guidelines for functions

- Remember the sum of squares example from a few slides ago
- We made a function called sum_of_squares
- sum_of_squares used a second function called square
- We broke the task down into two discrete functions
- We say that we **decomposed** the original problem
- We converted a complex problem into parts that are easier to conceive, understand, program, and maintain
- This makes our code robust, reusable, and scalable.

Functional decomposition

- Functional decomposition is the process of making sure our functions are:
  1. Short (the fewest lines possible)
  2. Atomic (cannot be broken down any further)
  3. General enough to be reused (modular)
  4. Understandable enough to require minimal comments
  5. Simple to test by themselves
- We should apply functional decomposition as we write our functions

DECOMPOSITION TECHNIQUES

An algorithm for decomposition

1. Start with a large function by identifying what it does (use a flowchart and identify the VERBS!)
2. Break VERBS off one at a time
3. You are trying to identify steps of the algorithm that are:
   a) Discrete (clear beginning and end)
   b) Repeated elsewhere in your code
   c) Useful or reusable/helpful to other parts of your codebase
4. Identify the arguments and the output to each portion (block) of code
   a) Arguments are the parameters
   b) Output is the return value
5. Extract the code and put it into its own function with a semantically meaningful name
6. Replace the point of extraction with a call to the new function, passing the required inputs and assigning the return value to a local variable.

How do we know we are done?

- That's an important question
- Different developers have different answers
- Generally desire short functions (no more than 10 or 15 lines, and shorter than that if possible)
- Each function:
  1. Does one logical thing
  2. Is highly cohesive (solves a small subproblem independently)
  3. Is loosely coupled to other functions, i.e., it does not need to know how other functions work
  4. Can be tested easily by itself.

How do we know we are done?

- All our functions are:
  - at (roughly) the same level of detail
  - easy to understand
- We can describe:
  - a function's main action with ONE simple verb
  - what a function does in a sentence or two
  - a function's inputs and output when it has been decomposed
- A well-decomposed function can be:
  - thoroughly tested to localize errors and minimize system faults
  - assembled with other functions to solve larger problems

if __name__ == '__main__'

if __name__ == '__main__'

- When a file (module) is imported, all code in the module is immediately executed
- Sometimes we don't want that, i.e., when we have a program spread over multiple files
- We want to way to tell the interpreter: *Hey interpreter, if this file is being used as a program, go ahead and read the file, but begin execution right here, and only execute this stuff right here!*

A ‘main’ function

- Many programming languages use a **main** function to do this
- *The main function is where the program begins*
- The runtime or interpreter executes the commands in the main function
- When we reach the end of the main function, the program ends
- *Every program can have many files, but only one main function*
- The Python approach is similar!

Where to start the program

- Python programs use the built-in variable __name__ to determine if the file is being executed as a program by the programmer, or if the file is being imported by another module
- If the value of __name__ is the string '__main__', then the file is executed as a program

```
def main( ):
    """Execute the program"""
    # Program starts here
    # Doing stuff
    # Doing more stuff
    # Program ends here

# Executes only if file run as a program
if __name__ == '__main__':
    main( )
```

That's it for week 03!

Any questions?