

# Clone Detection in Python

Valerio Maggio (valerio.maggio@unina.it)

Florence, Italy

**DATE:** May 13, 201<sup>3</sup>

### Number one in the stink parade is **duplicated code**.

If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.



DRIECT TECHNOLOG

## The Python Way

In [1]:	import this					
	The Zen of Python, by Tim Peters					
	Beautiful is better than ugly.					
	Explicit is better than implicit.					
1	Simple is better than complex.					
	Complex is better than complicated.					
	Flat is better than nested.					
	Sparse is better than dense.					
	Readability counts.					
	Special cases aren't special enough to break the rules.					
	Although practicality beats purity.					
	Errors should never pass silently.					
le l	Unless explicitly silenced.					
l'	In the face of ambiguity, refuse the temptation to guess.					
	There should be one and preferably only oneobvious way to do it.					
	Although that way may not be obvious at first unless you're Dutch.					
	Now is better than never.					
	Although never is often better than "right" now.					
	If the implementation is easy to explain, it is a bad idea.					
	Namospagos are one bonking great idea let's de more of these!					
	Namespaces are one nonking great idea iet's do more of thoset					

## The Python Way

In [1]:	import this
	The Zen of Python, by Tim Peters
	Beautiful is better than ugly.
8	Explicit is better than implicit.
	Simple is better than complex.
	Flat is better than nested.
	Sparse is better than dense.
1	Readability counts.
	Special cases aren't special enough to break the rules.
Freeze	Although practicality beats purity.
Errors	should never pass silenciy.
Unless	explicitly silenced.
In the	face of ambiguity, refuse the temptation to guess.
There s	hould be one and preferably only oneobvious way to do it
Althoug	h that way may not be obvious at first unless you're Dutch.
	Although never is orcen better than "right" now.
1	If the implementation is hard to explain, it's a bad idea.
	If the implementation is easy to explain, it may be a good idea.
	Namespaces are one honking great idea let's do more of those!

10	<pre>class ImmutableProbabilisticTree(ImmutableTree, ProbabilisticMixIn):</pre>	1.0	class ProbabilisticTree(Tree ProbabilisticMixIn):
20	<pre>defnew_(cls, node_or_str, children=None, **prob_kwaras);</pre>	20	def new (cls node or str children-None **nroh kwaras):
3 0	return super(ImmutableProbabilisticTree, cls), new (	3.0	return super(ProbabilisticTree cls) new (
4	cls, node or str. children)		cls node on str children)
5.0	def init (self, node or str. children=None, **prob kwaras):	5 0	def init (self node on str. children None **nroh kwanas);
6	if children is None: return # see note in Tree. init ()	5 M	if children is None: noture # see note in Treeinit ()
7	ImmutableTree, init (self, node or str. children)	7	Then init (solf node on str. children)
8	ProbabilisticMixIn, init (self, **prob kwaras)	0	ProhobilisticWixIn init (solf **nob kwanas)
9		0	ProbubilisticMixininit(sell, "prob_kwargs)
10	# We have to patch up these methods to make them work right:	10	# We have to natch up these methods to make them work night:
11 0	<pre>def frozen class(self): return ImmutableProbabilisticTree</pre>	11 0	def frozen class(self): return ImmutchleProbabilisticTree
12 0	<pre>defrepr(self):</pre>	12 0	def room (celf):
13	<pre>return '%s [%s]' % (Treerepr(self), self.prob())</pre>	13	$\frac{1}{2} = \frac{1}{2} = \frac{1}$
14 🕥	defstr(self):	14 0	def str (self):
15	<pre>return '%s [%s]' % (self.pprint(margin=60), self.prob())</pre>	15	return ' (self nnrint(margin=60)) self nroh())
16	<pre>defcmp(self, other):</pre>	16 0	def cmn (self other):
17	<pre>c = Treecmp(self, other)</pre>	17	c = Tree,  (self, other)
18	if c != 0: return c	18	if $c = 0$ : return $c$
19	<pre>return cmp(self.prob(), other.prob())</pre>	19	return cmp(self.prob(), other.prob())
20	<pre>defeq(self, other):</pre>	20 0	def eq (self, other):
21	if not isinstance(other, Tree): return False	21	if not isinstance(other, Tree): return False
22	return Treeeq(self, other) and self.prob()==other.prob()	22	return Tree, eq (self, other) and self, prob()==other.prob()
23 🕥	<pre>defne(self, other):</pre>	23 0	def ne (self, other):
24	return not (self == other)	24	return not (self == other)
25 🕥	<pre>def copy(self, deep=False):</pre>	25 🖸	<pre>def copy(self, deep=False):</pre>
26	<pre>if not deep: return selfclass(self.node, self, prob=self.prob())</pre>	26	if not deep: return self. class (self.node, self. prob=self.prob())
27	<pre>else: return selfclassconvert(self)</pre>	27	<pre>else: return selfclassconvert(self)</pre>
28 🖸	<pre>def convert(cls, val):</pre>	28	<pre>def convert(cls, val):</pre>
29	if isinstance(val, Tree):	29	if isinstance(val, Tree):
30	children = [cls.convert(child) for child in val]	30	children = [cls.convert(child) for child in val]
31	<pre>if isinstance(val, ProbabilisticMixIn):</pre>	31	if isinstance(val, ProbabilisticMixIn):
32	<pre>return cls(val.node, children, prob=val.prob())</pre>	32	<pre>return cls(val.node, children, prob=val.prob())</pre>
33	else:	33	else:
34	<pre>return cls(val.node, children, prob=1)</pre>	34	<pre>return cls(val.node, children, prob=1.0)</pre>
35	else:	35	else:
36	return val	36	return val
37	convert = classmethod(convert)	37	convert = classmethod(convert)
21	convert = classmethod(convert)	37	<pre>convert = classmethod(convert)</pre>
30	return val		return val
32	6[26]	32	else:
			a second second second with the second

1	class <u>ImmutableProbabilisticTree(ImmutableTre</u> e, ProbabilisticMixIn):	10	<pre>class ProbabilisticTree(Tree, ProbabilisticMixIn):</pre>
2 🕥	<pre>defnew(cls, node_or_str, children=None, **prob_kwargs):</pre>	2 🖸	<pre>defnew(cls, node_or_str, children=None, **prob_kwarqs):</pre>
3 🕥	<pre>return super(ImmutableProbabilisticTree, cls)new(</pre>	3 0	return super(ProbabilisticTree, cls)new(
4	cls, node_or_str, children)	4	cls, node_or_str, children)
5 🕥	<pre>definit(self, node_or_str, children=None, **prob_kwargs):</pre>	5 🕥	<pre>definit(self, node_or_str, children=None, **prob_kwaras);</pre>
6	if children is None: return # see note in Treeinit()	6	if children is None: return # see note in Tree. init_()
7	<pre>ImmutableTreeinit(self, node_or_str, children)</pre>	7	Tree, init (self, node or str. children)
8	ProbabilisticMixIninit(self, **prob_kwargs)	8	ProbabilisticMixIn, init (self, **prob kwaras)
9 🖸		9 🗖	······································
10	# We have to patch up these methods to make them work right:	10	# We have to patch up these methods to make them work right:
11 🕥	<pre>def _frozen_class(self): return ImmutableProbabilisticTree</pre>	11 0	def frozen class(self): return ImmutableProbabilisticTree
12	<pre>defrepr(self):</pre>	12 0	def repr (self):
13	<pre>return '%s [%s]' % (Treerepr(self), self.prob())</pre>	13	return '%s (n=%s)' % (Tree, repr (self), self.prob())
14 🕥	<pre>defstr(self):</pre>	14 🔾	def str (self):
15	<pre>return '%s [%s]' % (self.pprint(margin=60), self.prob())</pre>	15	return '%s (p=%s)' % (self.pprint(margin=60), self.prob())
16 🕥	<pre>defcmp(self, other):</pre>	16 🔾	def cmp (self, other):
17	<pre>c = Treecmp(self, other)</pre>	17	c = Tree, cmp (self, other)
18	if c != 0: return c	18	if c != 0: return c
19	<pre>return cmp(self.prob(), other.prob())</pre>	19	return cmp(self.prob(), other.prob())
20 🕥	<pre>defeq(self, other):</pre>	20 🔾	def eq (self, other):
21	if not isinstance(other, Tree): return False	21	if not isinstance(other, Tree); return False
22	<pre>return Treeeq(self, other) and self.prob()==other.prob()</pre>	22	return Tree, eq (self, other) and self, prob()==other, prob()
23 🕥	<pre>defne(self, other):</pre>	23 🔾	def ne (self, other):
24	return not (self == other)	24	return not (self == other)
25 🕥	<pre>def copy(self, deep=False):</pre>	25 🔾	def copy(self, deep=False):
26	<pre>if not deep: return selfclass(self.node, self, prob=self.prob())</pre>	26	if not deep: return self. class (self.node, self. prob=self.prob())
27	<pre>else: return selfclassconvert(self)</pre>	27	<pre>else: return self. class .convert(self)</pre>
28 🕥	<pre>def convert(cls, val):</pre>	28 🔾	def convert(cls, val):
29	if isinstance(val, Tree):	29	if isinstance(val. Tree):
30	children = [cls.convert(child) for child in val]	30	children = [cls.convert(child) for child in val]
31	<pre>if isinstance(val, ProbabilisticMixIn):</pre>	31	if isinstance(val. ProbabilisticMixIn):
32	<pre>return cls(val.node, children, prob=val.prob())</pre>	32	return cls(val.node, children, prob=val.prob())
33	else:	33	else:
34	<pre>return cls(val.node, children, prob=1)</pre>	34	return cls(val.node, children, prob=1.0)
35	else:	35	else:
36	return val	36	return val
37	convert = classmethod(convert)	37	convert = classmethod(convert)
21		37	convert = classmethod(convert)
32	convert - classmethod(convert)	30	return val
	return vol		else:
35	oleo.		

1 🗋	class <u>ImmutableProbabilisticTree(ImmutableTre</u> e, ProbabilisticMixIn):	1 🖸	<pre>class ProbabilisticTree(Tree, ProbabilisticMixIn):</pre>
2 🖸	<pre>defnew(cls, node_or_str, children=None, **prob_kwargs):</pre>	2 🖸	<pre>defnew(cls, node_or_str, children=None, **prob_kwargs):</pre>
3 🖸	<pre>return super(ImmutableProbabilisticTree, cls)new(</pre>	3 🕥	<pre>return super(ProbabilisticTree, cls)new(</pre>
4	cls, node_or_str, children)	4	<pre>cls, node_or_str, children)</pre>
5 🖸	<pre>definit(self, node_or_str, children=None, **prob_kwargs):</pre>	5 🕥	<pre>definit(self, node_or_str, children=None, **prob_kwargs):</pre>
6	<u>if children is None: return # see note in Treeinit()</u>	6	if children is None: return # see note in Treeinit()
7	<pre>ImmutableTreeinit(self, node_or_str, children)</pre>	7	Treeinit(self, node_or_str, children)
8	<pre>ProbabilisticMixIninit(self, **prob_kwargs)</pre>	8	ProbabilisticMixIninit(self, **prob_kwaras)
9 🖸		9 🗖	
10	# We have to patch up these methods to make them work right:	10	# We have to patch up these methods to make them work right:
11 🕥	<pre>def _frozen_class(self): return ImmutableProbabilisticTree</pre>	11.0	def frozen class(self): return ImmutableProbabilisticTree
12	<pre>defrepr(self):</pre>	12 0	def repr (self):
13	<pre>return '%s [%s]' % (Treerepr(self), self.prob())</pre>	13	return '%s (n=%s)' % (Tree, repr (self), self.prob())
14 🕥	<pre>defstr(self):</pre>	14 0	def str (self):
15	<pre>return '%s [%s]' % (self.pprint(margin=60), self.prob())</pre>	15	return '%s (n=%s)' % (self pprint( $margin=60$ ) self prob())
16	<pre>defcmp(self, other):</pre>	16 0	def cmp (self other):
17	<pre>c = Treecmp(self, other)</pre>	17	c = Tree  (self other)
18	if c != 0: return c	18	if c l= 0: return c
19	<pre>return cmp(self.prob(), other.prob())</pre>	10	return cmn(self nroh() other nroh())
20	<pre>defeq_(self, other):</pre>	20 0	def en (self other):
21	if not isinstance(other, Tree): return False	20 M	if not isingtance(other Tree); naturn False
22	return Tree. eq (self, other) and self.prob()==other.prob()	22	not use and (solf other) and solf prob()-other prob()
23 🔘	def ne (self. other):	22	def no (solf other):
24	return not (self == other)	20 M	derne(self, other).
25 0	<pre>def copy(self, deep=False):</pre>	25	def com/(colf_doon_Folco)
26	if not deep: return self. class (self.node, self. prob=self.prob())	25	der copy(setr, deep=raise):
27	else: return self. class .convert(self)	20	if not deep: return selfclass(self.hode, self, prod=self.prod())
28 0	def convert(cls, val):	20 0	else: return selfclassconvert(self)
29	if isinstance(val. Tree):	28	def convert(cls, val):
30	children = [c]s.convert(child) for child in val]	29	if isinstance(val, Tree):
31	if isinstance(val ProbabilisticMixIn):	30	children = [cls.convert(child) for child in val]
32	return cls(val node children prob-val prob())	31	if isinstance(val, ProbabilisticMixIn):
33	alse:	32	return cls(val.node, children, prob=val.prob())
34	return cls(val node children nrab-1)	33	else:
35	alse:	34	return cls(val.node, children, prob=1.0)
36	return vol	35	else:
37	convert - classrathed(convert)	36	return val
51		37	convert = classmethod(convert)
37	<pre>convert = classmethod(convert)</pre>	37	convert = classmethod(convert)
	return val		return val
32	else:	32	else:

1 🗋	<pre>class ImmutableProbabilisticTree(ImmutableTree, ProbabilisticMixIn):</pre>	1 🖸	<pre>class ProbabilisticTree(Tree, ProbabilisticMixIn):</pre>
2 🗋	<pre>defnew(cls, node_or_str, children=None, **prob_kwargs):</pre>	2 🕥	<pre>defnew(cls, node_or_str, children=None, **prob_kwargs):</pre>
3 🖸	<pre>return super(ImmutableProbabilisticTree, cls)new(</pre>	3 😭	<pre>return super(ProbabilisticTree, cls)new(</pre>
4	cls, node_or_str, children)	4	cls, node_or_str, children)
5 🖸	<pre>definit(self, node_or_str, children=None, **prob_kwargs):</pre>	5 🕥	<pre>definit(self, node_or_str, children=None, **prob_kwargs):</pre>
6	<u>if children is</u> None: return # see note in Treeinit()	6	if children is None: return # see note in Treeinit()
7	<pre>ImmutableTreeinit(self, node_or_str, children)</pre>	7	Treeinit(self, node_or_str, children)
8	<pre>ProbabilisticMixIninit(self, **prob_kwargs)</pre>	8	<pre>ProbabilisticMixIninit(self, **prob_kwargs)</pre>
9 🖸		9 🗖	
10	# We have to patch up these methods to make them work right:	10	# We have to patch up these methods to make them work right:
11 🕥	<pre>def _frozen_class(self): return ImmutableProbabilisticTree</pre>	11 0	<pre>def _frozen_class(self): return ImmutableProbabilisticTree</pre>
12 🕥	<pre>defrepr(self):</pre>	12 0	def repr (self):
13	<pre>return '%s [%s]' % (Treerepr(self), self.prob())</pre>	13	return '%s (p=%s)' % (Tree, repr (self), self,prob())
14 🕥	<pre>defstr(self):</pre>	14 0	def str (self):
15	<pre>return '%s [%s]' % (self.pprint(margin=60), self.prob())</pre>	15	return '%s (p=%s)' % (self.pprint(margin=60), self.prob())
16 🕥	<pre>defcmp(self, other):</pre>	16 0	def cmp (self, other):
17	<pre>c = Treecmp(self, other)</pre>	17	c = Tree  (self other)
18	if c != 0: return c	18	if c l= 0: return c
19	<pre>return cmp(self.prob(), other.prob())</pre>	19	return cmn(self nrob() other nrob())
20 🕥	<pre>defeq(self, other):</pre>	20 0	def en (self other):
21	if not isinstance(other, Tree): return False	20 1	if not isingtance(other Tree): return False
22	return Treeeq_(self, other) and self.prob()==other.prob()	22	return Tree and (self other) and self prob()-other prob()
23 🕥	<pre>defne_(self. other):</pre>	23	def no (salf other):
24	return not (self == other)	20 1	neturn net (self other)
25 🔿	<pre>def copy(self, deep=Fqlse);</pre>	25	def comu(colf_doon_Folco);
26	if not deep: return self. class (self.node, self. prob=self.prob())	25 M	if not door, noturn solf class (solf node solf prob solf prob())
27	else: return self. class .convert(self)	20	it not deep: return selfclass(self.node, self, prob=self.prob())
28 🔾	def convert(cls, val):	20 0	else: return seltclassconvert(selt)
29	if isinstance(val. Tree):	20	der convert(cls, val):
30	children = [cls.convert(child) for child in val]	29	if isinstance(val, Tree):
31	if isinstance(valProbabilisticMixIn):	30	children = [cls.convert(child) for child in val]
32	return cls(val node children proh-val proh())	31	if isinstance(val, ProbabilisticMixIn):
33	alse:	32	return cls(val.node, children, prob=val.prob())
34	return cls(val node, children, prah-1)	33	else:
35	also	34	return cls(val.node, children, prob=1.0)
36	etse.	35	else:
37	convent - classmethod(convent)	36	return val
57		37	convert = classmethod(convert)
35	convert = classmethod(convert)	37	<pre>convert = classmethod(convert)</pre>
	return val		return val
	6[26]	32	else:

1	<pre>class ImmutableProbabilisticTree(ImmutableTree, ProbabilisticMixIn):</pre>
2	<pre>defnew(cls, node_or_str, children=None, **prob_kwargs):</pre>
3 🖸	<pre>return super(ImmutableProbabilisticTree, cls)new(</pre>
4	cls, node_or_str, children)
5 🖸	<pre>definit(self, node_or_str, children=None, **prob_kwargs):</pre>
6	<pre>if children is None: return # see note in Treeinit()</pre>
7	<pre>ImmutableTreeinit(self, node_or_str, children)</pre>
8	<pre>ProbabilisticMixIninit(self, **prob_kwargs)</pre>
9 🖸	
10	# We have to patch up these methods to make them work right:
11 🖸	<pre>def _frozen_class(self): return ImmutableProbabilisticTree</pre>
12 🖸	<pre>defrepr(self):</pre>
13	<pre>return '%s [%s]' &amp; (Treerepr(self), self.prob())</pre>
14 🖸	<pre>defstr(self):</pre>
15	<pre>return '%s [%s]' % (self.pprint(margin=60), self.prob())</pre>
16 🖸	<pre>defcmp(self, other):</pre>
17	<pre>c = Treecmp(self, other)</pre>
18	if c != 0: return c
19	<pre>return cmp(self.prob(), other.prob())</pre>
20	<pre>defeq(self, other):</pre>
21	if not isinstance(other, Tree): return False
22	return Treeeq(self, other) and self.prob()==other.prob()
23	defne(self, other):
24	return not (self == other)
25	def copy(self, deep=False):
26	if not deep: return selfclass(self.node, self, prop=self.prop())
20	else: return seltclassconvert(self)
20	def convert( <i>cls</i> , <i>val</i> ):
29	children = [cls_convent(child) for child in val]
31	$if_i = [cis.convert(child) for child in val}$
32	return cls(val node children prob.val prob())
33	also
34	return cls(val node children prob_1)
35	alse.
36	return val
37	convert = classmethod(convert)
51	
37	<pre>convert = classmethod(convert)</pre>
	return val
32	else:

Ω	<pre>class ProbabilisticTree(Tree, ProbabilisticMixIn):</pre>
Ω	<pre>defnew(cls, node_or_str, children=None, **prob_kwargs):</pre>
	<pre>return super(ProbabilisticTree, cls)new(</pre>
	<pre>cls, node_or_str, children)</pre>
	<pre>definit(self, node_or_str, children=None, **prob_kwargs):</pre>
_	if children is None: return # see note in Treeinit()
	Treeinit(self, node_or_str, children)
	ProbabilisticMixIninit(self, **prob_kwargs)
	# We have to patch up these methods to make them work right:
	<pre>def _frozen_class(self): return ImmutableProbabilisticTree</pre>
	<pre>defrepr(self):</pre>
	<pre>return '%s (p=%s)' % (Treerepr(self), self.prob())</pre>
	<pre>defstr(self):</pre>
	<pre>return '%s (p=%s)' % (self.pprint(margin=60), self.prob())</pre>
	<pre>defcmp(self, other):</pre>
	<pre>c = Treecmp(self, other)</pre>
	if c != 0: return c
	<pre>return cmp(self.prob(), other.prob())</pre>
Ω	<pre>defeq(self, other):</pre>
	if not isinstance(other, Tree): return False
	<pre>return Treeeq(self, other) and self.prob()==other.prob()</pre>
	<pre>defne(self, other):</pre>
	return not (self == other)
Ω	<pre>def copy(self, deep=False):</pre>
	<pre>if not deep: return selfclass(self.node, self, prob=self.prob())</pre>
	<pre>else: return selfclassconvert(self)</pre>
Ω	<pre>def convert(cls, val):</pre>
	if isinstance(val, Tree):
	children = [cls.convert(child) for child in val]
	<pre>if isinstance(val, ProbabilisticMixIn):</pre>
	<pre>return cls(val.node, children, prob=val.prob())</pre>
	else:
	<pre>return cls(val.node, children, prob=1.0)</pre>
	else:
	return val
	convert = classmethod(convert)
	convert = classmethod(convert)
	return val
	6126:

return cls(val.node, children, prob

Introduction

1	class Immut	<u>ableProbabilisticTree(ImmutableTre</u> e, ProbabilisticMixIn):		10	clas
2	defr	<pre>new(cls, node_or_str, children=None, **prob_kwargs):</pre>		2 0	
3 🕥	ret	<pre>curn super(ImmutableProbabilisticTree, cls)new(</pre>		3 0	
4		cls, node_or_str, children)		4	
5 🕥	defi	<pre>nit(self, node_or_str, children=None, **prob_kwargs):</pre>		5.0	
6	if	children is None: return # see note in Treeinit()		6	
7	Imn	nutableTree,init(self, node_or_str, children)		7	
8	Pro	<pre>babilisticMixIninit(self, **prob_kwargs)</pre>		8	
9 🗖				9 11	
10	# We ho	ave to patch up these methods to make them work right:		10	
11 🕥	def _fr	<pre>rozen_class(self): return ImmutableProbabilisticTree</pre>		11.0	
12	defr	<pre>repr(self):</pre>		12 0	
13	ret	<pre>curn '%s [%s]' % (Treerepr(self), self.prob())</pre>		13	
14 🖸	defs	str(self):		14 0	
15	ret	<pre>curn '%s [%s]' % (self.pprint(margin=60), self.prob())</pre>		15	
16	def	<pre>mp(self, other):</pre>		16 0	
17	C =	Treecmp(self, other)		17	
18	if	c != 0: return c		18	
19	ret	<pre>curn cmp(self.prob(), other.prob())</pre>		19	
20 🕥	defe	eq(self, other):		20 0	
21	if	<pre>not isinstance(other, Tree): return False</pre>		21	
22	ret	<pre>curn Treeeq(self, other) and self.prob()==other.prob()</pre>		22	
23 🕥	defr	ne(self, other):		23 0	
24	ret	curn not (self == other)		24	
25 🕥	def cop	<pre>y(self, deep=False):</pre>		25 0	
26	if	<pre>not deep: return selfclass(self.node, self, prob=self.prob</pre>	()	26	
27	els	<pre>se: return selfclassconvert(self)</pre>		27	
28 🕥	def cor	<pre>vert(cls, val):</pre>		28 0	
29	if	isinstance(val, Tree):		29	
30		children = [cls.convert(child) for child in val]		30	
31		<pre>if isinstance(val, ProbabilisticMixIn):</pre>		31	
32		<pre>return cls(val.node, children, prob=val.prob())</pre>		32	
33		else:		33	
34		return cls(val.node, children, prob=1)		34	
35	els	se:		35	
36		return val		36	
37	convert	= classmethod(convert)		37	
				51	
37	convert	: = classmethod(convert)		37	
		return val			
32	els	G:		32	
34		<pre>return cls(val.node, children, prob=1)</pre>	Introd	lict:	ion
22				acc	

Ω	<pre>class ProbabilisticTree(Tree, ProbabilisticMixIn):</pre>
Ω	<pre>defnew(cls, node_or_str, children=None, **prob_kwargs):</pre>
0	<pre>return super(ProbabilisticTree, cls)new(</pre>
	cls, node_or_str, children)
	<pre>definit(self, node_or_str, children=None, **prob_kwaras);</pre>
_	if children is None: return # see note in Treeinit()
	Tree, init (self, node or str. children)
	ProbabilisticMixIn, init (self, **prob kwaras)
	······································
	# We have to patch up these methods to make them work right:
	<pre>def _frozen_class(self): return ImmutableProbabilisticTree</pre>
0	<pre>defrepr(self):</pre>
_	<pre>return '%s (p=%s)' % (Treerepr(self), self.prob())</pre>
	<pre>defstr(self):</pre>
	<pre>return '%s (p=%s)' % (self.pprint(margin=60), self.prob())</pre>
Ω	<pre>defcmp(self, other):</pre>
	<pre>c = Treecmp(self, other)</pre>
	if c != 0: return c
	<pre>return cmp(self.prob(), other.prob())</pre>
Ω	<pre>defeq(self, other):</pre>
	if not isinstance(other, Tree): return False
	<pre>return Treeeq(self, other) and self.prob()==other.prob()</pre>
Ω	<pre>defne(self, other):</pre>
	<pre>return not (self == other)</pre>
۵	<pre>def copy(self, deep=False):</pre>
	<pre>if not deep: return selfclass(self.node, self, prob=self.prob())</pre>
	<pre>else: return selfclassconvert(self)</pre>
۵	<pre>def convert(cls, val):</pre>
	if isinstance(val, Tree):
	children = [cls.convert(child) for child in val]
	<pre>if isinstance(val, ProbabilisticMixIn):</pre>
	<pre>return cls(val.node, children, prob=val.prob())</pre>
	else:
	<pre>return cls(val.node, children, prob=1.0)</pre>
	else:
	return val
	convert = classmethod(convert)
	convert = classmethod(convert)
	return val

4

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system
- Is often created during development

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system
- Is often created during development
  - due to time pressure for an upcoming deadline

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system
- Is often created during development
  - due to time pressure for an upcoming deadline
  - to overcome limitations of the programming language

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system
- Is often created during development
  - due to time pressure for an upcoming deadline
  - to overcome limitations of the programming language
- Three Public Enemies:

- Exists: 5% to 30% of code is similar
  - In extreme cases, even up to 50%
    - This is the case of Payroll, a COBOL system
- Is often created during development
  - due to time pressure for an upcoming deadline
  - to overcome limitations of the programming language
- Three Public Enemies:
  - Copy, Paste and Modify

# Clone Detection in Python

Part I: Clone Detection

**DATE:** May 13, 201<sup>2</sup>

# Clone Detection in Python

Part I: Clone Detection

**DATE:** May 13, 201<sup>2</sup>

## Code Clones

**(Def.)** "Software Clones are segments of code that are similar according to some definition of similarity" (I.D. Baxter, 1998)

- There can be different definitions of similarity, based on:
  - Program Text (text, syntax)
  - Semantics



## Code Clones

**(Def.)** "Software Clones are segments of code that are similar according to some definition of similarity" (I.D. Baxter, 1998)

- There can be different definitions of similarity, based on:
  - Program Text (text, syntax)
  - Semantics
- Four Different Types of Clones



```
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end----'):
    lines = list()
    with open(filepath) as report:
        for l in report:
            if l.endswith(marker):
                lines.append(l) # Stores only lines that ends with "marker"
        return lines #Return the list of different lines
```

## The original one

## Type 1: Exact Copy

 Identical code segments except for differences in layout, whitespace, and comments



## Type 1: Exact Copy

 Identical code segments except for differences in layout, whitespace, and comments

```
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
   lines = list()
   with open(filepath) as report:
      for 1 in report:
         if l.endswith(marker):
             lines.append(l) # Stores only lines that ends with "marker"
   return lines #Return the list of different lines
        def do_something_cool_in_Python (filepath, marker='---end---'):
           lines = list() # This list is initially empty
            with open(filepath) as report:
              for 1 in report: # It goes through the lines of the file
                  if l.endswith(marker):
                     lines.append(l)
           return lines
```

### Type 2: Parameter Substituted Clones

 Structurally identical segments except for differences in identifiers, literals, layout, whitespace, and comments



### Type 2: Parameter Substituted Clones

 Structurally identical segments except for differences in identifiers, literals, layout, whitespace, and comments

```
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
   lines = list()
   with open(filepath) as report:
      for 1 in report:
          if l.endswith(marker):
             lines.append(l) # Stores only lines that ends with "marker"
   return lines #Return the list of different lines
    # Type 2 Clone
    def do_something_cool_in_Python(path, end='---end---'):
       targets = list()
       with open(path) as data_file:
          for t in data file:
              if l.endswith(end):
                 targets.append(t) # Stores only lines that ends with "marker"
       #Return the list of different lines
       return targets
```



```
import os
def do_something_with(path, marker='---end---'):
  # Check if the input path corresponds to a file
   if not os.path.isfile(path):
      return None
   bad_ones = list()
   good_ones = list()
   with open(path) as report:
      for line in report:
         line = line.strip()
         if line.endswith(marker):
            good_ones.append(line)
         else:
            bad_ones.append(line)
   #Return the lists of different lines
   return good_ones, bad_ones
```

```
import os
def do_something_with(path, marker='---end---'):
  # Check if the input path corresponds to a file
   if not os.path.isfile(path):
      return None
   bad_ones = list()
   good_ones = list()
   with open(path) as report:
      for line in report:
         line = line.strip()
         if line.endswith(marker):
            good_ones.append(line)
         else:
            bad_ones.append(line)
   #Return the lists of different lines
   return good_ones, bad_ones
```

```
import os
def do_something_with(path, marker='---end---'):
  # Check if the input path corresponds to a file
   if not os.path.isfile(path):
      return None
   bad_ones = list()
   good_ones = list()
   with open(path) as report:
      for line in report:
         line = line.strip()
         if line.endswith(marker):
            good_ones.append(line)
         else:
            bad_ones.append(line)
   #Return the lists of different lines
   return good_ones, bad_ones
```

```
import os
def do_something_with(path, marker='---end---'):
  # Check if the input path corresponds to a file
   if not os.path.isfile(path):
      return None
   bad_ones = list()
   good_ones = list()
   with open(path) as report:
      for line in report:
         line = line.strip()
         if line.endswith(marker):
            good_ones.append(line)
         else:
            bad_ones.append(line)
   #Return the lists of different lines
   return good_ones, bad_ones
```

## Type 4: "Semantic" Clones

Semantically equivalent segments that perform the same computation but are implemented by different syntactic variants



## Type 4: "Semantic" Clones

 Semantically equivalent segments that perform the same computation but are implemented by different syntactic variants

```
# Original Fragment
def do_something_cool_in_Python(filepath, marker='---end---'):
    lines = list()
    with open(filepath) as report:
        for l in report:
            if l.endswith(marker):
                lines.append(l) # Stores only lines that ends with "marker"
        return lines #Return the list of different lines
```

```
def do_always_the_same_stuff(filepath, marker='---end---'):
    report = open(filepath)
    file_lines = report.readlines()
    report.close()
    #Filters only the lines ending with marker
    return filter(lambda l: len(l) and l.endswith(marker), file_lines)
```

### What are the consequences?

- Do clones increase the maintenance effort?
- *Hypothesis*:
  - Cloned code increases code size
  - A fix to a clone must be applied to all similar fragments
  - Bugs are duplicated together with their clones
- However: it is not always possible to <u>remove</u> clones
  - Removal of Clones is harder if variations exist.

Clc	one Do	etectior	ר To	ols	
Du	uploc				
	SDD	Dup	CPD	6-0	
NICAD	Dude	Duplix	Gemir	ni	
Simian	CLICS	CCFinder	Shino	bi	
		Clone Detectiv	e	iClones	
Sco	orpio	ConQAT	Clor	ne Digger	
Duplix		Decka	Deckard		
-	PIVID	CloneDr K	Clone	SimScan	



## Clone Detection Tools

Duploc

SDD NiCAD Dude

Simian

CLICS

Scorpio

Duplix

Dup<br/>CPDDuplixGeminiCCFinderShinobiClone DetectiveiClones

### • Token Based Tools:

 Token sequences are compared to sequences

#### **Clone Detection Tools** Duploc Dup CPD SDD **NiCAD** Duplix Gemini Dude Simian **CCFinder** Shinobi **CLICS Clone Detective** iClones onQAT Syntax Based Tools: **Clone Digger** Deckard Syntax subtrees are JCCD compared to each other bneDr SimScan **KClone**

## Clone Detection Tools

Duploc

NiCAD Dude Simian

CLICS

DupCPDDuplixGeminiCCFinderShinobiClone DetectiveiClones

Scorpio Duplix PMD

### Graph Based Tools:

(sub) graphs are compared to each other

## **Clone Detection Techniques**

- String/Token based Techiniques:
  - Pros: Run very fast
  - Cons: Too many false clones
- **Syntax** based (AST) Techniques:
  - Pros: Well suited to detect structural similarities
  - Cons: Not Properly suited to detect Type 3 Clones
- **Graph** based Techniques:
  - Pros: The only one able to deal with **Type 4 Clones**
  - Cons: Performance Issues

### The idea: Use Machine Learning, Luke



- Use Machine Learning Techniques to compute similarity of fragments by exploiting specific *features* of the code.
- Combine different sources of Information
  - Structural Information: **ASTs**, **PDGs**
  - Lexical Information: Program Text

### Kernel Methods for Structured Data

- Well-grounded on solid and awful Math
- Based on the idea that objects can be described in terms of their constituent Parts
- Can be easily tailored to specific domains
  - Tree Kernels
  - Graph Kernels
  - ...



The definition of a new Kernel for a **Structured Object** requires the definition of:

Set of **features** to annotate each part of the object

- Set of **features** to annotate each part of the object
- A Kernel function to measure the similarity on the smallest part of the object

- Set of **features** to annotate each part of the object
- A Kernel function to measure the similarity on the smallest part of the object
  - e.g., Nodes for AST and Graphs

- Set of **features** to annotate each part of the object
- A Kernel function to measure the similarity on the smallest part of the object
  - e.g., Nodes for AST and Graphs
  - A Kernel function to apply the computation on the different (sub)parts of the structured object

### Kernel Methods for Clones: Tree Kernels Example on AST

- Features: We annotate each node by a set of 4 *features* 
  - Instruction Class
    - i.e., LOOP, CONDITIONAL\_STATEMENT, CALL
  - Instruction
    - i.e., FOR, IF, WHILE, RETURN
  - Context

FOR

- i.e. Instruction Class of the closer statement node
- Lexemes
- Lexical information gathered (recursively) from leaves
- i.e., Lexical Information



# Clone Detection in Python

Part II: Clones and Python

**DATE:** May 13, 201<sup>2</sup>

# Clone Detection in Python

Part II: Clones and Python

**DATE:** May 13, 2013



## The Overall Process Sketch



## The Overall Process Sketch



### The Overall Process Sketch







### **Detection Process**

## **Empirical Evaluation**

- Comparison with another (pure) AST-based: Clone Digger
  - It has been the first Clone detector for and in Python :-)
  - Presented at EuroPython 2006
- Comparison on a system with randomly seeded clones

### Results refer only to Type 3 Clones

On Type 1 and Type 2 we got the same results



**Precision:** How **accurate** are the obtained results? (Altern.) How many errors do they contain?





### Is Python less clone prone?

## Clones in CPython 2.5.1

#### atic PyObject \*

betime\_new(PyTypeObject \*type, PyObject \*args, PyObject \*kw)

```
PyObject *self = NULL;
PyObject *state;
int year;
int month;
int day;
int hour = 0;
int minute = 0;
int second = 0;
int usecond = 0;
PyObject *tzinfo = Py_None;
```

#### /\* Check for invocation from pickle with \_\_getstate\_\_ state \*/

```
if (PyTuple_GET_SIZE(args) >= 1 &&
    PyTuple_GET_SIZE(args) <= 2 &&
    PyString_Check(state = PyTuple_GET_ITEM(args, 0)) &&
    PyString_GET_SIZE(state) == _PyDateTime_DATETIME_DATASIZE &&
    MONTH_IS_SANE(PyString_AS_STRING(state)[2]))
```

```
PyDateTime_DateTime *me;
char aware;
```

3

```
recurn
```

```
memcpy(me->data, pdata, _PyDateTime_DATETIME_DATASIZE);
me->hashcode = -1;
me->hastzinfo = aware;
if (aware) {
        Py_INCREF(tzinfo);
        me->tzinfo = tzinfo;
```

#### static PyObject \* time\_new(PyTypeObject \*type, PyObject \*args, PyObject \*kw)

```
PyObject *self = NULL;
PyObject *state;
int hour = 0;
int minute = 0;
int second = 0;
int usecond = 0;
PyObject *tzinfo = Py_None;
```

#### /\* Check for invocation from pickle with \_\_getstate\_\_ state \*/

```
if (PyTuple_GET_SIZE(args) >= 1 &&
    PyTuple_GET_SIZE(args) <= 2 &&
    PyString_Check(state = PyTuple_GET_ITEM(args, 0)) &&
    PyString_GET_SIZE(state) == _PyDateTime_TIME_DATASIZE &&
    ((unsigned char) (PyString_AS_STRING(state)[0])) < 24)
```

PyDateTime\_Time \*me; char aware;

3

if (PyTuple\_GET\_SIZE(args) == 2) {
 tzinfo = PyTuple\_GET\_ITEM(args, 1);
 if (check\_tzinfo\_subclass(tzinfo) < 0) {
 PyErr\_SetString(PyExc\_TypeError, "bad "
 "tzinfo state arg");
 }
}</pre>

#### return NULL;

```
return (PyObject *)me;
```



# Thank you!

Florence, Italy

**DATE:** May 13, 201<sup>2</sup>